Introduction

The Siren system is a general-purpose music composition and production framework integrated with Squeak Smalltalk (1); it is a Smalltalk class library of about 200 classes for building musical applications. Siren runs on a variety of platforms with support for real-time MIDI and multi-channel audio I/O. The system's source code is available for free on the Internet; see the Siren home page at the URL http://www.create.ucsb.edu/Siren.

This chapter concentrates on (a) the Smoke music description language, (b) the real-time MIDI and sound I/O facilities, and (c) the GUIs for the 2.7 version of Siren. It is intended for a Squeak programmer who is interested in music and sound applications, or for a computer music enthusiast who is interested in Squeak applications.

The motivation for the development of Siren is to build a powerful, flexible, and portable computer-based composer's tool and instrument. It should support composition and realization of electroacoustic music with abstract notations and structure representation, sound and score processing. The goal of the high-level and application software is to exhibit good object-oriented design principles and elegant modern software engineering practice.

There are several elements to Siren:

- a general-purpose music representation system (music magnitudes, events, event lists, generators, functions, and sounds);
- I/O voices, schedulers and I/O drivers (real-time and file-based voices for sound and MIDI);
- user interface components for musical applications (UI framework, widgets, and tools); and
- a collection of built-in applications (editors and browsers for Siren objects).

The figure below shows the basic packages that comprise Siren. At the center are the class categories that make up the Smoke music and multimedia representation language. On the left side are the editors and applications for manipulating Smoke objects in Siren. To the right are the various objects that handle input-output in various formats and real-time drivers.
Music and Sound Processing Using Siren

Siren Architecture

(2 alternatives – which do the reviewers prefer?)

Siren and its direct hardware and software predecessors stem from music systems that developed in the process of my composition. The first generation of object-oriented music software I used, called ARA, was an outgrowth of the Lisp system wrote for the composition of *Bat out of Hell* (1983); the DoubleTalk system was based on the Smalltalk-80-based Petri net editing system used for *Requiem Aeternam dona Eis* (1986) (2); the HyperScore ToolKit’s various versions were used (among others) for *Day* (1988) (3). Siren’s direct predecessor, the MODE, was used for *Kombination XI* (1990/98) and *All Gates Are Open* (1993/95) (4).

In each of these cases, some amount of effort was spent-after the completion of the composition—to make the tools more general-purpose, often making them less useful for any particular task. Siren (5, 6)—a re-implementation of the MODE undertaken in 1997-9—is based on the representations and tools used in the realization of *Ywe Ye, Yi Jr Di* (1999/2000). The "clean-up" effort was minimized here; the new package is much more useful for a much smaller set of tasks and attitudes about what music representation and composition are. If Siren works well for other composers, it is because of its idiosyncratic approach, rather than its attempted generality (i.e., the instrument approach rather than the tool approach).
Music and Sound Processing Using Siren


Software for Music Experimentation

In the computer music literature, the primary programming languages used for experimental systems have been Lisp and Smalltalk; this can be traced to several basic concepts. Both languages provide an extremely simple, single-paradigm programming model (i.e., all data are of the same basic “type” and all behavior is accessed in the same way) and consistent syntax that scales well to large expressions (this is a matter of debate among “language bigots”). 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 and Smalltalk-based operating systems demonstrates the scalability of Lisp and Smalltalk 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.

Two important language features that are common to both Lisp and Smalltalk are dynamic typing and dynamic polymorphism. Dynamic typing means that data type information is specific to (run-time) values and not (compile-time) variables as in many other languages. In Pascal or C, for example, one declares all variables as typed (e.g., int i; means that variable i is an integer), and may not generally assign other kinds of data to a variable after its declaration (e.g., i = "hello"; trying to assign a string to i). Declaring a variable name in Lisp or Smalltalk says nothing about the types of values that may be assigned to that variable. While this generally implies some additional run-time overhead, dynamic binding is a valuable language asset because of the increase it brings in software flexibility, abstraction and reusability.

Polymorphism means being able to use the same function name with different types of arguments to evoke different behaviors. Most languages allow for some polymorphism in the form of overloading of their arithmetical operators, meaning that one can say 3 + 4 or 3.1 + 4.1 in order to add integers or floating-point numbers. The problem with limited 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 Lisp and Smalltalk, all functions can be overloaded, so that one can create many types of objects that can be used interchangeably (e.g., many different types of objects can handle the play message in their own ways). Using polymorphism also incurs a run-time overhead, but, as with dynamic binding, it can be considered essential for a language on which to base an exploratory programming environment for music and multimedia applications.
There is a rich and diverse literature related to the representation, manipulation, and interchange of multimedia data in general, and musical sound and scores in particular. Two useful surveys are those by Roger Dannenberg () and Gerrant Wiggins, et al. ()

Among the important issues are (1) which media related to sound and music are to be supported: recorded sound, musical performance, musical structure; (2) what level of sonic semantics and musical structure is to be supported, and (3) how exactly the representation is to capture an actual performance or recording. In many systems, issue (1) is addressed by a small and fixed set of data types (e.g., sound-only, control-only, or events-only), and a trade-off is seen between issues (2) and (3)–between what Wiggins et al. call structural generality and representational completeness.

Many object models for complex domains start with objects that represent the basic “units of measure” or magnitudes of the domain. In the case of sound and music, this means classes to model the basis properties of sounds and musical events such as time (and/or duration), meter, pitch, loudness, and spatial dimensions. Along with some model of generic “events” (at the level of words or musical notes), one must build micro-level functions and control objects, and higher-level “event lists” to represent sentences, melodies, and other composite events. This basic design is very similar to the patterns found in graphics systems based on display lists.

The Smoke Music Representation Language

The "kernel" of Siren is the set of classes for music magnitudes, functions and sounds, events, event lists and event structures known as the Smallmusic Object Kernel (Smoke) music representation. Smoke is described in terms of two related description languages (music input languages), a compact binary interchange format, and concrete data structures. The high-level packages of Siren—voices, sound/DSP, compositional structures, and the user interface framework—interoperate using Smoke event lists.

Smoke supports the following levels of description:

• abstract models of the basic musical quantities (scalar magnitudes such as duration, pitch, loudness or duration);

• instrument/note (voice/event or performer/score) pattern for mapping abstract event properties onto concrete parameters of output media or synthesis methods;

• functions of time, sampled sound, 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 description levels;

• nested/hierarchical event-tree structures for flexible description of "parts," "tracks," or other parallel/sequential organizations;
• separation of "data" from "interpretation" (what vs. how in terms of providing for interpretation objects called voices);

• abstractions for the description of "middle-level" musical structures (e.g., chords, clusters, or trills);

• annotation of event tree structures supporting the creation of heterarchies (lattices) and hypermedia networks;

• annotation including graphical data necessary for common-practice notation; and

• description of sampled sound synthesis and processing models such as sound file mixing or DSP.

Given a flexible and abstract basic object model for Smoke, it should be easy to build converters for many common formats, such as MIDI data, formatted note lists, DSP code, or mixing scripts, and it should be possible to parse live performance data (e.g., incoming MIDI streams) into Smoke, and to interpret it (in some rendition) in real-time.

The "executive summary" of Smoke from (Smoke 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 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.

Smoke objects also have behaviors for managing several special types of links, which are seen simply as properties where the property name is a symbol such as usedToBe, isTonalAnswerTo, or obeysRubato, and the property value is another Smoke object, e.g., an event list. With this facility, one can built multimedia hypermedia navigators for arbitrary Smoke networks. The three example link names shown above could be used to implement event lists with version history, to embed analytical information in scores, or to attach real-time performance controllers to event lists, respectively.
Music and Sound Processing Using Siren

Music Magnitudes

MusicMagnitude objects are characterized by their identity, class, species, and value (e.g., the pitch object that represents ‘c3’ has its object identity, the class SymbolicPitch, the species Pitch, and the value ‘c3’ [a string]). MusicMagnitude behaviors distinguish between class membership and species in a multiple-inheritance-like scheme that allows the object representing “440.0 Hz” to have pitch-like and limited-precision-real-number-like behaviors. This means that its behavior can depend on what it represents (a pitch), or how its value is stored (a floating-point number).

The mixed-mode music magnitude arithmetic is defined using the technique of species-based coercion, i.e., class Pitch knows whether a note name or Hertz value is more general. This provides capabilities similar to those of systems that use the techniques of multiple inheritance and multiple polymorphism (such as C++ and the Common Lisp Object System), but in a much simpler and scalable manner. All meaningful coercion messages—e.g., (440.0 Hz) asMIDIKeyNumber—and mixed-mode operations—e.g., (1/4 beat + 80 msec)—are defined.

The basic model classes include Pitch, Loudness, and Duration; exemplary extensions include Length, Sharpness, Weight, and Breath for composition- or notation-specific magnitudes. The handling of time as a parameter is finessed via the abstraction of duration. All times are durations of events or delays, so that no “real” or “absolute” time object is needed. Duration objects can have simple numerical or symbolic values, or they can be conditions (e.g., the duration until some event X occurs). Boolean expressions of other durations, or arbitrary blocks of Smalltalk-80 code.

Functions of one or more variables are yet another type of signal-like music magnitude. The Function class hierarchy includes line segment, exponential segment, spline segment and Fourier summation functions.

In the verbose Smoke format music magnitudes, events and event lists are created by instance creation messages sent to the appropriate classes. The first three expressions in the examples below create various music magnitudes and coerce them into other representations.

The terse form for music magnitude creation uses post-operators (unary messages) such as 440 hz or 250 msec, as shown in the subsequent examples.

Users can extend the music magnitude framework with their own classes that refine the existing models of define totally new kinds of musical metrics.

Verbose MusicMagnitude Creation and Coercion Messages

(Duration value: 1/16) asMsec “Answers Duration 62 msec.”
(Pitch value: 60) asHertz “Answers Pitch 261.623 Hz.”
(Amplitude value: ‘ff’) asMIDI “Answers MIDI key velocity 100.”

Mixed-mode Arithmetic

[(1/2 beat) + 100 msec] “(0.6 beat)”
['a4' pitch + 25 Hz] “(465.0 Hz)”
['a4' pitch + 100 Hz] asMIDI “(73 key)”
['mp' ampl + 3 dB] “(-4.6 dB)”
Music and Sound Processing Using Siren

Events and Event Lists

The **AbstractEvent** object in Smoke is modeled as a property-list dictionary with a duration. Events have no notion of external time until their durations become active. Event behaviors include duration and property accessing, and “performance,” where the semantics of the operation depends on another object—a voice or driver as described below.

The primary messages that events understand are: *(anEvent duration: someDurationObject)*—to set the duration time of the event (to some Duration-type music magnitude)—and property accessing messages such as *(anEvent color: #blue)*—to set the “color” (an arbitrary property) to an arbitrary value (the symbol #blue).

The meaning of an event’s properties is interpreted by voices and user interface objects; it is obvious that (e.g.,) a pitch could be mapped differently by a MIDI output voice and a graphical notation editor. It is common to have events with complex objects as properties (e.g., envelope functions, real-time controller maps, DSP scripts, structural annotation, version history, or compositional algorithms), or with more than one copy of some properties (e.g., one event with enharmonic pitch name, key number, and frequency, each of which may be interpreted differently by various voices or structure accessors).

That there is no prescribed “level” or “grain size” for events in Smoke. There may be a one-to-one or many-to-one relationship between events and “notes,” or single event objects may be used to represent long complex textures or surfaces.

Note the way that Smoke uses the Smalltalk concatenation message “,” to denote the construction of events and event lists; *(magnitude, magnitude)* means to build an event with the two magnitudes as properties, and *(event, event)* or *((duration -> event), (duration -> event))* means to build an event list with the given events as components.

There are classes for events as follows.

- **AbstractEvent** – just a property list
- **DurationEvent** – adds duration
- **MusicEvent** – adds pitch and voice
- **ActionEvent** – has a block that it evaluates when scheduled

It is seldom necessary to extend the hierarchy of events.

**Verbose Event Creation Messages – Class messages**

“Create a ‘generic’ event.”

MusicEvent duration: 1/4 pitch: ‘c3’ ampl: ‘mf’

“Create one with added properties.”

(MusicEvent dur: 1/4 pitch: ‘c3’) color: #green; accent: #sfz

**Terse Event Creation using concatenation of music magnitudes**

- [440 Hz, (1/4 beat), 44 dB]
- [490 Hz, (1/7 beat), 56 dB, (#voice -> #flute),]
EventList objects hold onto collections of events that are tagged and sorted by their start times (represented as the duration between the start time of the event list and that of the event). The event list classes are subclasses of DurationEvent themselves. This means that event lists can behave like events and can therefore be arbitrarily deeply nested, i.e., one event list can contain another as one of its events.

The primary messages to which event lists respond (in addition to the behavior they inherit by being events), are (anEventList add: anEvent at: aDuration)–to add an event to the list–(anEventList play)–to play the event list on its voice (or a default one)–(anEventList edit)–to open a graphical editor in the event list–and Smalltalk-80 collection iteration and enumeration messages such as (anEventList select: [someBlock])–to select the events that satisfy the given (Boolean) function block.

Event lists can map their own properties onto their events in several ways. Properties can be defined as lazy or eager, to signify whether they map themselves when created (eagerly) or when the event list is performed (lazily). This makes it easy to create several event lists that have copies of the same events and map their own properties onto the events at performance time under interactive control. Voices handle mapping of event list properties via event modifiers, as described below.

In a typical hierarchical Smoke score, data structure composition is used to manage the large number of events, event generators and event modifiers necessary to describe a full performance. The score is a tree–possibly a forest (i.e., with multiple roots) or a lattice (i.e., with cross-branch links between the inner nodes)–of hierarchical event lists representing sections, parts, tracks, phrases, chords, or whatever abstractions the user desires to define. Smoke does not define any fixed event list subclasses for these types; they are all various compositions of parallel or sequential event lists.

Note that events do not know their start times; this is always relative to some outer scope. This means that events can be shared among many event lists, the extreme case being an entire composition where one event is shared and mapped by many different event lists (as described in [Scaletti 1989]). The fact that the Smoke text-based event and event list description format consists of executable Smalltalk-80 message expressions (see examples below), means that it can be seen as either a declarative or a procedural description language. The goal is to provide “something of a cross between a music notation and a programming language” (Dannenberg 1993).

The verbose way of creating an event list is to create a named instance and add events explicitly as shown in the first example below, which creates a D-major chord.

```
[(EventList newNamed: #Chord1)
  add: (1/2 beat, 'd3' pitch, 'mf' ampl) at: 0;
  add: (1/2 beat, 'fs3' pitch, 'mf' ampl) at: 0;
  add: (1/2 beat, 'a4' pitch, 'mf' ampl) at: 0]
```

This same chord could be defined more tersely as,
Music and Sound Processing Using Siren

[(0 => (1/2 beat, 'd3' pitch, 'mf' ampl)),
 (0 => (1/2 beat, 'fs3' pitch, 'mf' ampl)),
 (0 => (1/2 beat, 'a4' pitch, 'mf' ampl))]

This could be done even more compactly using a Chord object (see the discussion of event generators below) as,

[[Chord majorTriadOn: 'd3' inversion: 0] eventList]

Terse EventList creation using concatenation of events or (duration, event) associations looks like,

[(440 Hz, (1/2 beat), 44.7 dB),
 "note the comma between events"
 (1 => ((1.396 sec, 0.714 ampl) phoneme: #xu))]

Siren I/O

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. There are voice “device drivers” for common file storage formats—such as cmusic note lists, the Adagio language, MIDI file format, or phase vocoder scripts—or for use with real-time schedulers connected to MIDI or sampled sound drivers.

These classes can be refined to add new event and signal file formats or multilevel mapping (e.g., for MIDI system exclusive messages) in an abstract way. **Voice** objects can also read input streams (e.g., real-time controller data or output from a coprocess), and send messages to other voices, schedulers, event modifiers or event generators. This is how one uses the system for real-time control of complex structures.

Some voices are “timeless” (e.g., MIDI file readers); they operate at full speed regardless of the relative time of the event list they read or write. Others assume that some scheduler hands events to their voices in real time during performance. The EventScheduler does just this; it can be used to sequence and synchronize event lists that may include a variety of voices.

Examples

Create a random event list and write it out to a cmix-format notelist file. Edit the file.

[CmixVoice randomExampleToFileNamed: ‘rand.cm’]

Create a random event list and write it out to a cmusic-format notelist file. Edit the file.

[CmusicVoice randomExampleToFileNamed: ‘rand.sc’]

Test playing Siren event lists using JohnM’s synthesized voices

[SynthVoice default play: (EventList randomExample: 20)]

Read MIDI, turn it into a Siren event list, and play it.

[((MIDIFileReader scoreFromFileNamed: ‘BWV775.MID’) asEventList) play]
Music and Sound Processing Using Siren

Real-time music I/O in Siren is managed by Squeak primitive interfaces to sound and MIDI OS-level drivers. The glue code for these primitives is written in Smalltalk and translated to C for linking with the Squeak virtual machine (itself written in Smalltalk and translated). Several sets of primitives exist for Squeak on various platforms, including support for sound synthesis, digital audio signal processing, MIDI event-oriented and continuous controller I/O, and VM-level scheduling.

User interfaces for music/sound processing

Navigator MVC in Siren

The Smalltalk-80 Model-View-Controller (MVC) user interface paradigm (2), is well-known and widely imitated. The traditional three-part MVC architecture involves a model object representing the state and behavior of the domain model—in our case, an event list or signal. The view object presents the state of the model on the display, and the controller object sends messages to the model and/or the view in response to user input.

“Navigator MVC” (4) (see the figure) is a factoring of the controller/editor and view for higher levels of reuse. The fundamental feature of this architecture is that all applications are built as display list editors (i.e., the generic tool is “smart draw”), with special layout manager objects for translating the model structure into a graphical display list representation and for translating structure interaction into model manipulation.

A StructureAccessor is an object that acts as a translator or protocol converter. An example might be an object that responds to the typical messages of a tree node or member of a hierarchy (e.g., What's your name? Do you have and children/sub-nodes? Who are they? Add this child to them.). One specific, concrete subclass of this might know how to apply that language to navigate through a hierarchical event list (by querying the event list's hierarchy). The role of the LayoutManager object is central to building Navigator MVC applications. Siren's layout manager objects can take data structures (like event lists) and create display lists for time-sequential (i.e., time running left-to-right or top-to-bottom), hierarchical (i.e., indented list or tree-like), network or graph (e.g., transition diagram), or other layout formats. The editor role of Navigator MVC is played by a smaller number of very generic (and therefore reusable) objects such as EventListEditor or SampledSoundEditor, which are shared by most of the applications in the system.
Music and Sound Processing Using Siren

Figure: Navigator MVC Architecture

Much of the work of building a new tool within the Siren often goes into customizing the interaction and manipulation mechanisms, rather than just the layout of standard pluggable view components. Building a new notation by customizing a layout manager class and (optionally) a view and controller, is relatively easy. Adding new structure accessors to present new perspectives of structures based on properties or link types can be used to extend the range of applications and to construct new hypermedia link navigators. This architecture means that views and controllers are extremely generic (applications are modeled as structured graphics editors), and that the bulk of many applications' special functionality resides in a small number of changes to existing accessor and layout manager classes.

The Siren implementation of Navigator MVC is integrated with the Morphic (3) graphics framework.

Siren MVC Examples

The example screens below show the simple Siren display list editor (“SqueakDraw”); it allows you to manipulate hierarchical structured graphics objects. The pop-up menu in the left view shows the default display list controller message. Keyboard commands and mouse interaction support zooming and scrolling. One item is selected in each view, and can be dragged or resized using its “selection handles.”

Figure: Siren DisplayList Examples
Music and Sound Processing Using Siren

The example below a shows a class inheritance hierarchy presented as an indented list. Color is used to denote class species relationships in the class hierarchies. A refined tree layout manager could do graphical balancing or top-down layout.

Figure: Siren LayoutManager Example

A time sequence view is a display list view that whose layout manager interprets time as running from left to right. In the example below, the note “head” is used to display the event’s voice; this is a steady rhythm of 4 voices.

Figure: Siren TimeSequenceView Example

In a pitch/time view, time runs left-to-right, and pitch is displayed from bottom-to-top. In the example below, the layout manager creates a separate sub-display-list for each note, adding lines to the note head image to show its amplitude (vertical in red) and duration (horizontal in blue). Hauer-Steffens view is related to piano-roll notation; the lines are placed corresponding to the accidentals or the black notes on the keyboard.
Figure: Siren Score Editor Example

The multi-function view allows the viewing and editing of up to 4 functions of 1 variable. The example shows linear break-point functions in red and yellow, an exponential segment function in blue, and a cubic spline function in green. The buttons along the left are for selecting a particular function for editing or file I/O.

Figure: Siren Function Editor Example

The sonogram view displays an FFT-derived spectrum. In the example below shows a swept sine wave.

Figure: Siren Sonogram Example
Examples of Siren applications

**Sound/music databases: Paleo**

Most prior work in sound or music databases has addressed a single kind of data (e.g., MIDI scores or sampled sound effects), and has pre-defined the types of queries that are to be supported (e.g., queries on fixed sound properties or musical features). Earlier systems also tended to address the needs of music librarians and musicologists, rather than composers and performers. In the Paleo system under development at CREATE since 1996, we have built a suite of sound and music analysis tools that is integrated with an object-oriented persistency mechanism and a rapid application development environment.

The central architectural feature of Paleo is its use of dynamic feature vectors and on-demand indexing, whereby annotational information derived from data analysis can be added to items in the database at any time, and where users can develop new analysis or querying techniques and then have them applied to the database's contents on-the-fly within a query. For data that is assumed to be musical sound, this might mean performing envelope detection, spectral analysis, linear prediction, physical model parameter estimation, transient modeling, etc. For musical performance data (e.g., MIDI), this might entail extraction of expressive timing, phrase analysis, or harmonic analysis.

Paleo content is assumed to be sampled sound, musical scores, or captured musical performances. Scores and performance formats can be simple (e.g., MIDI-derived) or may contain complex annotation and embedded analysis. Paleo is specifically constructed to support multiple sets of captured musical performances (for use in comparing performance expression). This includes the derivation of basic timing and dynamics information from MIDI performances (to be able to separate the performance from the “dead-pan” score), and the analysis of timbral information from recorded sounds.

For score analysis, we use a variety of methods, including simple statistical models, rule-based analysis, and constraint derivation. Sampled sound analysis is undertaken using a suite of functions called NOLib that are written in the MatLab language and can be accessed from within the Paleo environment over the net via socket-based MatLab servers. The techniques available in NOLib include all standard time-, frequency-, wavelet modulus-domain analysis operations, as well as pitch detection, instrument classification, and sound segmentation.

The two main applications for Paleo are its use in an integrated tool-kit for composers, and in a performer's rehearsal workstation. The first set of applications will put the database at the core of a composition development environment that includes tools for thematic and sonic experimentation and sketch data management. Over the course of 1999 Paleo was integrated into an existing composer's software framework called the “Interim DynaPiano.” The second platform centers on manipulating rehearsal performance data relative to a “reference” score (which may or may not be a “dead-pan” interpretation). Users can play into the system, and then compare their performance to another one of their own or of their teacher's. Query preparation takes place using pre-built tools such as the composer's
Music and Sound Processing Using Siren

sketch browser, or by creating direct queries in a simplified declarative query language.

The implementation of the Paleo database persistency and access component is based on the public-domain Minnstore object-oriented database package (Carlson 1998), which allows flexible management of data and indices. The Squeak port of Minnstore is called SMS (Squeak Minnstore).

Paleo applications can communicate with an SMS database server over a network, and can pass sound sample data or event streams to/from the database. We currently use a simple socket-based protocol, but plan to move to a CORBA-based distribution infrastructure in the near future.

To stress-test Paleo’s analysis and query tools against a realistic-sized data set, the test contents used at the start of 2000 included over 1000 scores of keyboard music (Scarlatti, Bach, Bartok, the hymnal, etc.), several hundred “world” rhythms, the SHARC database of instrument tone analyses, 100 recorded guitar performance techniques, flute performances, and spoken poetry in five languages.

Paleo Architecture

In Paleo, as in Siren, music and sound data are represented via Smoke objects. In Paleo’s SMS data persistency layer, Smoke objects are stored in object sets, which are akin to database tables. Each object set stores one kind of objects, and can have any number of stored or derived indices. The collection of all defined indices determines the feature vector of the object set. When stored to disk, each object set has its own directory, storage policy, and a group of index files. For performance reasons, there are also cache policies per object set, and methods exist for keeping active object sets in a RAM disk.

Various services can be used by the SMS database server, such as call-outs to the NOLib functions (see below) or the use of extra Smalltalk processes for data analysis. The SMS server really only provides the persistency layer and cache policies for open object sets. The overall architecture is as shown in Figure below.

Figure: Paleo Architecture

The NOLib Analysis Functions

NOLib is a suite of data analysis and feature extraction routines written by our colleague Nicola Orio in the MatLab programming language. These
functions can be called by analysis scripts (interpreted MatLab programs), which can themselves be started by a network-based “analysis server.” We use the public-domain “octave” (Octave 1999) implementation of MatLab running on SGI (IRIX) and Apple (LinuxPPC) servers.

MIDI File Analysis with Constraints

Our purpose is to allow complex queries on various kinds of musical data, including scores, in the spirit of the Humdrum system (Huron 1994). A large amount of digitized music is available as MIDI files, for instance on one of the MIDI archives on the Internet.

The MIDI format however, provides only low-level musical information: it is rather a performance- than an analysis-oriented representation of music. Thus, we need to analyze MIDI files in order to compute additional musical features, such as: pitch-classes (by resolving enharmonic ambiguities), voice leading, keys, and harmonies.

The different tasks of analysis — enharmonic, melodic, tonal, and harmonic analysis — are not independent. For instance, the enharmonic analysis depends on tonal analysis, and conversely, the computation of local keys is based on the frequency of the different pitch-classes. Therefore, we need a global strategy in which the different tasks are performed simultaneously.

In our context, we often need a partial analysis because many queries only involve a few specific elements or incomplete information. Consider the following queries: “How many sonatas by Scarlatti end with a perfect cadenza?” or “Are there more minor than major chords in the preludes of Bach’s WTC?” In such cases, it is useless to perform a complete harmonic analysis of the 555 sonatas by Scarlatti, or of the 48 preludes of the WTC. This speaks for a scheme allowing partial and incomplete analysis.

What to analyze also depends on various parameters, such as the epoch, the style, and the nature (i.e. form, instrumentation) of the music considered, e.g. the anatomic limitations of human voice compared to a keyboard instrument. Our analysis strategy should be easily adaptable to various situations.

The previous remarks led us to considering an approach based on constraint satisfaction, instead of using specific algorithms for the different tasks on analysis (Mouton 1994). First, as a declarative paradigm, constraint satisfaction permits to build systems that can be adapted to specific situations easily. For instance, adapting the system to vocal or keyboard music analysis is just a matter of using a different set of constraints for the melodies. Besides, constraint resolution can be partial and incomplete. More precisely, the query “How many sonatas by Scarlatti end with a perfect cadence?” will only require the computation of elements related to the last two chords of each sonata. Finally, constraint resolution is a global process, in which the different elements are progressively computed, thus, interdependent tasks are interlaced in the resolution.

A constraint satisfaction problem or CSP (Mackworth 1977) consists of a set of variables (each one associated with a set of possible values, its domain), representing the unknown values of the problem, and a set of
Music and Sound Processing Using Siren

constraints, expressing relationships between them. Solving a CSP consists in instantiating each variable with a value in its domain so that the constraints are satisfied.

Our approach to analyzing a MIDI file consists in the following steps. First, we quantify the MIDI file in order to get rid of slight tempo fluctuations, and we segment it into a series of positions. Then, we define a CSP, whose variables represent the different elements of analysis: notes (one for each MIDI note-event), chords (at each position), keys (at each position), and melodies, and whose constraints represent the relationships holding between them. The set of constraints depends on the style and the form of the piece. Then we solve the CSP using standard CSP resolution. We use the BackTalk (Roy 1998) constraint solver to state and solve the problem.

Paleo I/O Formats

Paleo supports compact and efficient data I/O in the form of methods that work with the Squeak Smalltalk ReferenceStream framework, a customizable binary object streaming format. The trade-offs in the design of object storage formats are between size, complexity, and flexibility (pick any two). In Paleo, we opted for a system that is compact but also supports the full flexibility of the Smoke music representation, including abstract models for pitch, time, and dynamics, multiple levels of properties and annotation, the attachment of functions of time to events, and hyper-links between events or event lists.

Below is a hexadecimal dump of a simple event list (score) that demonstrates mixed-type properties, i.e., the pitches are mixed among MIDI key numbers, note names, and Hertz values. The format shown here is the “debugging format” and has some extra key bytes and explicit class names rather than more compact symbol table keys.

<table>
<thead>
<tr>
<th>Hex data bytes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>09 00 00 00 06</td>
<td>Object header</td>
</tr>
<tr>
<td>key 09 = normal object</td>
<td></td>
</tr>
<tr>
<td>32-bit inst size 6 = 4 instVars + 1 property + 1 offset</td>
<td></td>
</tr>
<tr>
<td>06 09 45 76 65 6E 74 4C 69 73 74</td>
<td></td>
</tr>
<tr>
<td>Object type: class name, key 06 = String, size = 09, value = 'EventList'</td>
<td></td>
</tr>
<tr>
<td>80 58 04 00 00 08 6E</td>
<td></td>
</tr>
<tr>
<td>Event list header, properties (dur) key 80 = music magnitude, key 58 = MSecondDuration, key 04 = smallInt, 32-bit int value = 08 6E msec</td>
<td></td>
</tr>
<tr>
<td>01 01 01</td>
<td></td>
</tr>
<tr>
<td>nil pitch, ampl, voice</td>
<td></td>
</tr>
<tr>
<td>06 04 6E 61 6D 65</td>
<td>property key = 'name'</td>
</tr>
</tbody>
</table>
Music and Sound Processing Using Siren

06 03 74 65 36  property value = 'te6'
00 00 00 0D  # of events (13)

End of header, note data follows

C0 80 58 04 00 00 00 00

Note event header, C0 = event key,
start time, 80 = music mag. key
58 = msec dur key, 04 = smallInt,
32-bit val = 0

00 00 00 05  obj size = 4 instVars + 0 props + 1
06 0A 4D 75 73 69 63 45 76 65 74

class name, key 06 = String,
size = 0A, = 'MusicEvent'

80 58 04 00 00 00 A6

duration, key 58 = MSec dur,
key 04 = smallInt,
32-bit int value A6 = 166 msec

80 5C 11 02 63 32

pitch, key 5c = SymbolicPitch,
key 11 = string,
length = 2, string = 'c2'

80 47 04 00 00 00 48

loudness, key 47 = MIDIVelocity
voice = nil, End of event

C0 80 58 04 00 00 00 A6

Event header, (start time) music
magnitude, msec dur

00 00 00 05  obj size = 4 iVars + 0 props + 1
0A 00 00 00 35  reference to class name
80 48 0E 3F C5 3F 7C

duration, key 48 = SecondDuration,
key 0E = float

80 5C 11 03 63 23 32

pitch, key 5c = SymbolicPitch,
key 11 = string...

80 54 0E C0 13 B7 BB

loudness, key 47 = DBLoudness,
key 0E = float

01  voice = nil, End of event

—more events follow

Data files in this format is on the order of 10-40 times larger than the “corresponding” MIDI files, but because this notation supports the full Smoke annotation, we can store much richer data. Paleo extensions include simple derived properties such as symbolic pitch (with enharmonic
Music and Sound Processing Using Siren

disambiguation) and time (with tempo and meter derivation, rest insertion, and metrical grouping), and higher-level properties such as harmonic analysis, performance expression, and others.

Using Paleo

To set up Paleo, we create a database given a storage directory, then create one or more object sets in it (these correspond to classes or tables), and lastly define indices for the object sets (corresponding to instance variables and accessors). One can then add object to an object set, or retrieve objects based on queries.

Create a new database of scores

The first example establishes a new database and adds an object set to it. The object we add to this set are assumed to respond to the messages composer and style. The examples that follow are in Smalltalk; comments are enclosed in double-quotes.

```
| dir db |
dir := 'Nomad:Paleo'.               "base directory"
db := SMSDB newOn: dir.
   "DB object"
(db addObjectSetNamed: #Scores)
   "Add an obj-set"
   objectsPerFile: 1;
   "Stores event lists"
   storesClass: EventList;
   "Add 2 indices"
   indexOn: #composer domain: String;
   indexOn: #style domain: Symbol.
db save.               "Save the object set"
db storeAll: (...collection_of_scores...)
```

Make a simple query

To make a simple database query, we re-open the database, and create a getOne: message with one or more where: clauses, e.g., to get a score by name.

```
| db |
db := MinneStoreDB openOn: 'Nomad:Paleo'.
(db getOne: #Scores)         "Create a query on name"
   where: #name eq: #ScarlattiK004;
   execute            "Get the first response"
```

Add a new index to an existing database

To add a new index to an existing object set, we use the indexOn: message, giving it the name of a "getter" method (i.e., the method that answers the property of the index), or simply provide a block of Smalltalk code to execute to derive the index value. In the second part of the next example, we create an index of the pitches of the first notes in the score database using a block (the text between the square brackets) that gets the
first pitches. This getter block could involve more complex code and/or calls to NOLib functions.

“Add a new index with getter method”

(db objectSetNamed: #Scores)
  indexOn: #name domain: Symbol.

“Add an index with getter block”

(db objectSetNamed: #Scores)
  indexOn: #firstPitch
domain: SmallInteger
  getter: [:el | el events first event pitch asMIDI value].

Make a more sophisticated query

To retrieve objects from the database, we use getOne: or getAll: as above, and can, for example, ask for a range or the derived first-pitch feature.

(db getAll: #Scores)
  where: #firstPitch between: 62 and: 65;
  execute

The “Tschinn Bum” Opera Database

Research has been underway at CREATE since 1996 on the topic of flexible frameworks for sound and music databases. In 1999, we published the first papers and source code releases of the Paleo music/sound database system, written in the Smalltalk programming language and based on the Smoke object-oriented music representation language.

Shortly thereafter, we started a collaboration with our colleague Professor Patricia Hall, who is leading the efforts to preserve and disseminate the known sketches of the opera “Wozzeck” by Alban Berg. Professor Hall has collected high-quality photographs of all known sources (sketches, particell, and full score manuscripts), and has scanned much of this material into computer image files. The task presented itself of how to package these images for easy browsing and use by musicologists, music theorists, students, and other users.

Using the “Tschinn Bum!” prototype, a user can access abstract information in the database about the musical and text structure of the scene, bibliographic data related to the images, and other criteria for generating queries that search the image database. The “Tschinn Bum!” system is novel because it requires both a high-level abstract music representation system (so that one can ask questions about the musical structure) with a large-scale image storage framework (since the images add up to several hundred megabytes).

…more here…
REFERENCES (formatting to come)


Octave Programming Language. See http://www.che.wisc.edu/octave/


Music and Sound Processing Using Siren
