Stephen Travis Pope

The Nomad Group
P.O. Box 60632
Palo Alto, California 94306 USA
stp@CCRMA.Stanford.edu

The Interim DynaPiano: An Integrated Computer Tool and Instrument for Composers

The *Interim DynaPiano* (IDP) is an integrated computer hardware and software configuration for music composition, production, and performance based on a Sun Microsystems Inc. SPARCstation computer and the Musical Object Development Environment (MODE) software. The IDP SPARCstation is a powerful hardware-accelerated color graphics workstation computer, based on a reduced instruction set computer (RISC) and running the UNIX operating system. It is augmented by large random access and disk memories and coprocessors and interfaces for real-time sampled sound and MIDI I/O. The MODE is a large hierarchy of object-oriented software components for music written in the Smalltalk-80 language and programming system. MODE software applications in IDP support flexible structured music composition, sampled sound recording and processing, and real-time music performance using MIDI or sampled sounds.

The motivation for the development of IDP is to build a powerful, flexible, and portable computer-based composer's tool and musical instrument that is affordable for a professional composer (i.e., approximately the price of a good piano or MIDI studio). The hardware and low-level software of the system consist entirely of off-the-shelf commercial components. The goal of the high-level and application software is to exhibit good object-oriented design principles and elegant modern software engineering practice. The basic configuration of the system is consistent with a whole series of "intelligent composer's assistants" based on a core technology that has been stable for a decade.

This article presents an overview of the hardware and software components of the current IDP system.

Computer Music Journal, Vol. 16, No. 3, Fall 1992,

The background section discusses several of the design issues in IDP in terms of definitions and a set of examples from the literature. The hardware system configuration is presented next, and the rest of the article is a description of the MODE signal and event representations, software libraries, and application examples.

Background

The name *Interim DynaPiano* was chosen for this system for historical reasons. One of the first publications describing the Smalltalk system under development at the Xerox Palo Alto Research Center (PARC) in the 1970s was *Personal Dynamic Media* (LRG 1976), a report that described the Smalltalk-76 system developed by Alan Kay, Adele Goldberg, and others, running on a Xerox Alto personal computer. The introduction of the report describes in some detail the hardware and software for a "DynaBook" system—a personal dynamic medium for learning and creating—which was unfortunately not implementable at the time. They include a photograph of a mock-up of a "future DynaBook," something that looks astonishingly like a current "notebook" or "tablet" personal computer. Because "Alto" was a Xerox-internal code name, the authors were not allowed to use it, so they presented the system they had developed as an *Interim DynaBook* machine (the Alto PC), with the "interim" software environment Smalltalk-76. Several sound and music applications are included in the examples they present. In the spirit that each machine and each software system is "interim," it was decided to dub the system described here *Interim DynaPiano* because it is the first computer-based musical tool/instrument I've built or used that provides a comfortable program-

ming environment and has adequate real-time signal *and* event processing capabilities to be used both as a stand-alone production studio, and as a real-time performance instrument.

The IDP and its direct hardware and software predecessors stem from music systems that developed in the process of my composition. Of the MODE's ancestors, *ARA* was an outgrowth of the Lisp system used for *Bat out of Hell* (1983); *DoubleTalk* was the Smalltalk-80-based Petri net editing system used for *Requiem Aeternam dona Eis* (1986); and the *HyperScore ToolKit's* various versions were used (among others) for *Day* (1988). 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. The MODE—a reimplementation of the HyperScore ToolKit undertaken in 1990–91—is based on the representations and tools used in the realization of *Kombination XI* (1990). 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 the IDP with MODE works well for other composers, it is because of its idiosyncratic approach, rather than its attempted generality.

There are two other trade-offs that determine the configuration and usage of a system like the IDP: ease-of-learning versus power and flexibility; and real-time performance versus power. Much modern software is designed to be extremely easy to learn, to the detriment of its power and flexibility. The Apple Macintosh vs. UNIX debate is a prime example of this—the Macintosh can be seen as a computer for children (or inexperienced or infrequent users), and a UNIX machine as one for "consenting adults." IDP with the MODE software, in particular, is a large and complex system that is nontrivial to learn to use effectively. The hope is that this is more than made up by the power and flexibility of the system. The system is designed to give the best possible real-time performance, but not to be limited by the amount of processing that is possible in real time. The system should provide guaranteed real-time performance for simple tasks (such as MIDI I/O), and should degrade gracefully when real-time processing cannot be achieved.

The IDP Literature

The development of interactive workstation-based computer music systems started shortly after personal computers with enough power and flexibility to support musical applications became available. By the mid-1970s, several teams had already demonstrated integrated hardware and software tools for real-time software sound synthesis or synthesizer control (e.g., on Alto computers at Xerox PARC, or on early Lisp Machines at the MIT AI Lab).

Workstation-Based Computer Music Systems

The design and configuration of the IDP system described here has been influenced by several streams of computer music systems. The core technologies of the hardware and software components of IDP are very similar to those used in a number of other computer music workstation systems, both current and in the literature of the last decade (and to my first effort, the Cadmus 9230/m workstation demonstrated at the 1984 International Computer Music Conference in Paris). These components are a commercial engineering workstation (e.g., 680X0- or RISC-based), large RAM and soundfile disk memories, a multitasking operating system (e.g., UNIX), real-time sampled sound I/O and/or MIDI I/O drivers and interfaces, C-based music and DSP software libraries and language (e.g., CARL/cmusic), and a flexible, interactive, graphical software development and delivery environment (e.g., Lisp or Smalltalk).

The importance of using a commercial PC or workstation computer is simply wide availability—IDP should not require an electrical engineer to configure it. A powerful multitasking operating system with built-in lower-level signal- and event-handling drivers and support libraries is required so that the higher-level software components can be flexible, portable, and abstract. These high-level and front-end components must be written in an interpreted or rapid-turnaround, incrementally-compiled software development and delivery environment. The use of a powerful and abstract central programming language integrated with the user interface "shell" is very important to the "dyna" part of an IDP; the goal is to

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, Shrobe, and Sandewall 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 (1) a powerful, abstract programming language with automatic storage management, interpreted and/or incrementally compiled execution, and a run-time available-compiler; (2) software libraries including reusable low- and high-level modules; (3) a windowed interactive user interface "shell" text and/or menu system; (4) a set of development, debugging, and code management tools; (5) an interface to "foreign-language" (often C and assembler) function calls; and (6) a software framework for constructing interactive graphical applications.

The primary languages for such systems have been Lisp and Smalltalk-80 (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 large 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 Objectworks/Smalltalk system from ParcPlace Systems, Inc., is a sophisticated development environment that is also portable, fast, commercially supported, and stable. Other commercial Smalltalk systems (e.g., from Digitalk) or public domain systems (e.g., GNU) are largely source-code compatible with "standard" Smalltalk-80.

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 *values* and not *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. Although 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.

IDP-like Systems of the 1980s

There has been varying progress in the evolution of each of the hardware and software components listed above between the systems developed in 1982 and

1983 and those used today, such as IDP. Comments on several of the systems that have influenced the current design will precede the detailed presentation of the IDP. The research systems that appeared along the coasts of the United States in the mid-to-late 1970s were generally programmed in languages that were not mainstream and used noncommercial (i.e., not widely available) hardware. The advent of UNIX and the Motorola 68000 microprocessor led to a plethora of commercial UNIX workstation computers in the first years of the 1980s that went by the moniker "3M machines"—with 1 million instructions per second (MIPS) CPU performance, 1 Mbyte RAM memory, and 1 megapixel bitmapped screen resolution. Several groups used these to develop *computer music workstations* or *intelligent composer's assistants.*

If I were allowed to write "Interactive Composition Systems I Have Known and Loved," it would have to open with a discussion of the SSSP synthesizer developed by Bill Buxton and others at the University of Toronto in the late 1970s. SSSP combined a Digital Equipment Corporation, DEC PDP-11 computer with special user interface and sound synthesis hardware (Buxton et al. 1978). The synthesizer could produce up to 16 voices using various synthesis methods in real time under the control of the PDP-11. The software was a broad range of UNIX-based C-language routines and applications that all manipulated the same event and voice data structures (Buxton et al. 1979).

The software distribution put together in 1982 by F. Richard Moore and D. Gareth Loy at the Computer Audio Research Laboratory (CARL) at the University of California in San Diego (UCSD) included comprehensive C language libraries for event and signal processing of all sorts, and the *cmusic* "sound compiler," a simple, extensible member of the Music-V family. The tools are integrated and used via the UNIX C-shell, C preprocessor, and C compiler. The simplicity, comprehensiveness, and extensibility of the CARL software has made it the basis of several powerful computer music research and production tools on a variety of platforms, including DEC VAX, Sun workstations, and NeXT machines.

The *Cadmus 9230/m* (for music) workstation configuration developed by Günther Gertheiss, Ursula Linder, and the author at PCS/Cadmus computers in Munich in 1983 and 1984 was based on an asymmetrical multiprocessor with MC68010 CPUs for the operating system, the window system, the Ethernet driver, the file system, and other dedicated uses (Cadmus 1985). The DAC/ADC system was based on a (then new) Sony converter evaluation card with a parallel QBus interface and buffer card. The system had 2–4 Mbytes of RAM and 220 Mbytes of disk memory in the default configuration and cost $35,000.

Three software environments were implemented on the system; the first was based on my earlier PDP-11-based *mshell* (Pope 1982), an interactive event and signal processing language based on the C-shell and simple windowing, graphics, and menu interaction functions. Mshell was a front-end for creating and editing function and note-list files for the *Music11* non-real-time sound synthesis system (using *cmusic* and much better graphics on the Cadmus machine), and provided many synthesis and DSP functions as basic language operators. The second software platform, ARA, was based on FranzLisp, the *orbit* object-oriented language, and a Lisp foreign-function call interface to the functions of the CARL software libraries. Starting in late 1984, a series of Smalltalk-80 music languages and tools were built for the 9230/m, leading to DoubleTalk and the HyperScore ToolKit in 1986.

Christopher Fry's *Flavors Band* (Fry 1984) was a tool kit based on the Symbolics Lisp Machine and the notion of phrase processors that are manipulated and applied in a menu-oriented and Lisp-listener user interface. It used a pre-MIDI connection to a Fender Rhodes Chroma synthesizer for output and was used for a variety of styles and productions. Flavors Band phrase processors served as the models for several current interaction paradigms, such as Miller Puckette's Max and the MODE's event generators and event modifiers.

The *CHANT/FORMES* environment developed at IRCAM between 1980 and 1985 by Rodet and Cointe (1984) used (ObjV)Lisp running (in various incarnations) on DEC VAX, Sun workstation, and Apple Macintosh computers. The basic description and processing systems of the FORMES environment were extended by Macintosh-based graphical user inter-

faces, connection to the CHANT synthesis language, and the *Esquisse* composition system.

Kyma is a graphical sound manipulation and composition language developed by Carla Scaletti (1989). It is linked to the *Capybara,* a scalable real-time digital signal processor built by Kurt Hebel. The Smalltalk-80-based software tools that Kyma comprises present a uniquely abstract and concrete composition and realization platform. Kyma is one of the only full IDP systems (as defined above) delivered on a personal computer (i.e., PC or Macintosh); it is also a prime example of multilanguage integration with Smalltalk-80; in it Smalltalk methods generate, download, and schedule microcode for the Capybara DSP, as well as reading and writing MIDI.

The recent *Common Lisp music/Common music* (clm/cm) system developed by William Schottstaedt and Heinrich Taube at the Center for Computer Research in Music and Acoustics (CCRMA) at Stanford University combines a NeXT "cube" workstation with an Ariel Corporation *Quint Processor* with five Motorola DSP56001 coprocessors. The combined system supports signal synthesis and processing, score description and management, and MIDI capture and performance in a unified Lisp-based environment. This is a powerful state-of-the-art Lisp-based music system.

The *IRCAM Musical Workstation* by Eric Lindemann, Miller Puckette and others (Lindemann et al. 1991) uses a Next "cube" with a card designed at IRCAM and manufactured by Ariel that contains two Intel i860 floating-point signal processors and a Motorola DSP56001 for I/O. The higher-level software includes tools for performance (e.g., Max) and programming (e.g., Animal). The signal processing capabilities of the system are impressive, but a reasonably configured system will have a price more in the range of a Bösendorfer than an IDP, and the software tools are far from stable.

Several nonexamples deserve citation to further demonstrate the definition of IDP. The various C/C++/Objective C-based systems such as that of the NeXT computer's Sound and Music Kits, the "standard" CARL environment, or the Composer's Desktop Project do provide sound compilers, vocoders, and sound processing tools, but fail to integrate them into a fully interactive exploratory programming environment (see again Deutsch and Taft 1980). The range of Macintosh-based MIDI packages includes flexible programmable systems for music (e.g., MIDI-Lisp or HMSL), but these address only the event and event-list levels (often using MIDI note events as the only abstraction), and generally rely on MIDI's crude model of signals.

Many other PC- and workstation-based signal-oriented systems generally fall into the categories of closed applications (e.g., the Studer/Editech *Dyaxis,* hard-disk recorders, *Music-N* compiler tool kits, or samplers), which are not programmable "composer's workbenches." Such applications must be provided in an open and customizable way for an IDP to be "dyna."

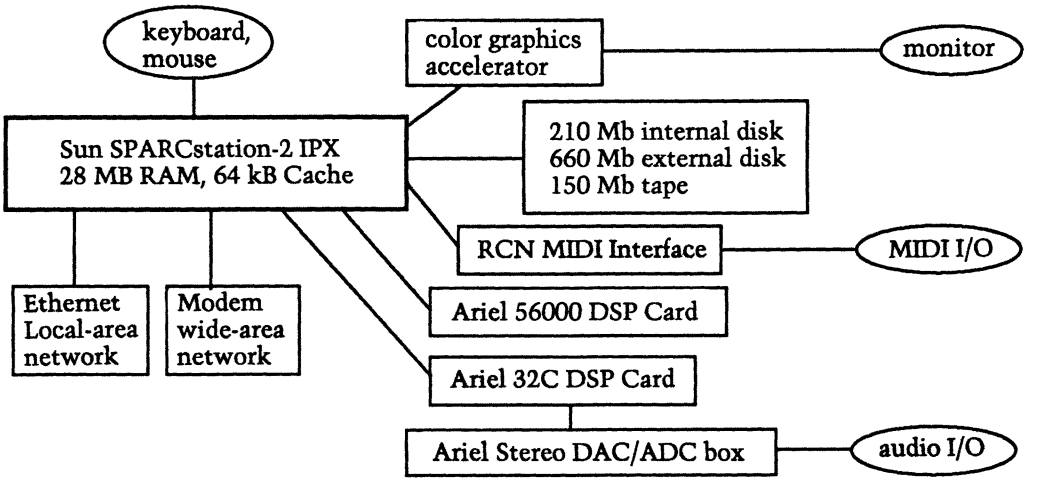
IDP Hardware Configuration

The IDP is based entirely on off-the-shelf commercial hardware and low-level software for the simple reason that the author refuses ever again to wire-wrap a DAC or to debug a play program. Figure 1 shows the configuration of the current (Spring 1992) mobile IDP system. The general-purpose and audio- or MIDI-specific components and specifications are listed below.

Sun Microsystems Inc. SPARCstation-2 IPX workstation, with
  28.5 MIPS, 4.2 MFLOPS, 24.2 SPECmarks CPU performance;
  28 Mbyte RAM, 64 kb cache memory;
  210 + 660 Mbyte SCSI disk memory;
  streamer tape cartridge;
  Ethernet (local-area network) interface;
  color graphics accelerator;
  16-inch Sony color monitor, keyboard, mouse;
  internal CODEC (telephone-quality) DAC/ADC; and Telebit high-speed modem (wide-area network).

Ariel Corp. S56X SBus card with
  Motorola DSP56001 24-bit fixed-point coprocessor;
  64 kbyte local RAM memory;
  SBus DMA controller;
  programmable Xilinx coprocessor; and a
  NeXT-compatible (DSP56000 SSI/SCI) DSP-Port connector.

Fig. 1. Current IDP hardware configuration.



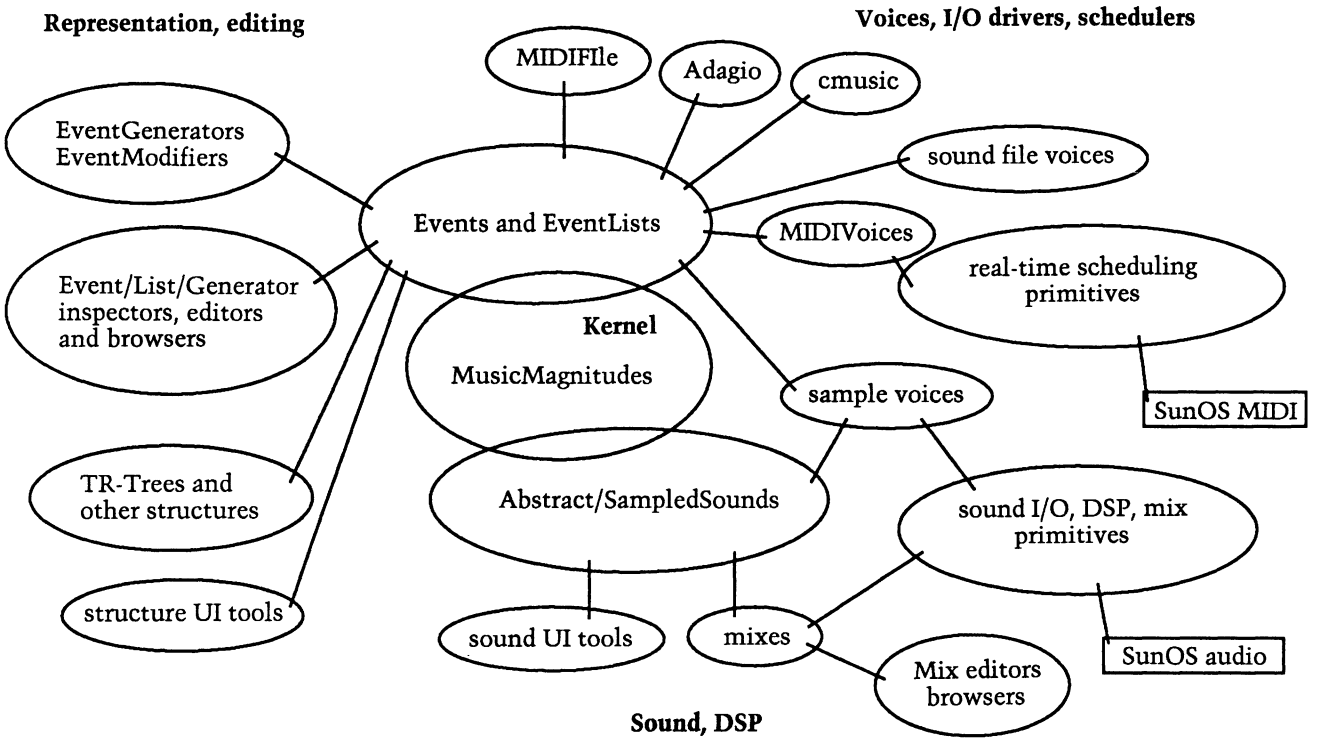
Ariel Corp. S-32C SBus card, with
AT&T DSP32C 32-bit floating-point coprocessor;
256 kbyte local RAM memory;
DSP-Port connector.
Ariel Corp. ProPort outboard DAC/ADC, with
pro-audio-quality 44.1 or 48 kHz stereo I/O; and
balanced analog audio I/O connectors.
RCN GmbH MIDI I/O interface, with
MIDI inputs and outputs from RS232 serial line.

The "core" hardware system consists of the SPARCstation, one of the Ariel DSP boards, and the ProPort converter box. The second Ariel board, the MIDI interface, the modem, etc., are all nonessential (but nice to have). The two DSP coprocessors are used for different reasons. The DSP32C-based board is faster, supports floating-point arithmetic (a requirement for such applications as vocoders), and has better software development tools (the AT&T C compiler). The DSP56000-based card allows for faster I/O because of its DMA controller, and can be configured to support two ProPorts (via the Xilinx part), providing for four-channel sampled sound I/O (which has been demonstrated at a 44.1 kHz sample rate). The large RAM (28–40 MBytes) is chosen in order to support multiple memory-hungry programs (e.g., X/OpenWindows and Smalltalk-80), and still have large in-core sound buffers. This allows near-real-time mixing of multiple soundfiles and simple sampler operation. Planned future expansion (aside from getting ever larger disks), includes the addition of a STEIM SensorLab analog control signal preprocessor, so that alternative MIDI controllers can be built that use the IDP as a synthesizer.

The main difference between this system and the older ones mentioned above is quantitative. The CPU power and memory capacities have increased roughly eight- to tenfold relative, for example, to the Cadmus system of 8 years ago, and the dedicated DSP and scheduling facilities have been dramatically improved. The hardware system is now small enough to travel (in two flight cases that are approximately 80 cm cubes), is powerful enough to execute sophisticated real-time applications, and costs less (around $20,000 at present). The analog audio I/O is also of good enough quality for professional performance or recording.

There are other hardware systems that would allow such a system to be configured, most notably the new Silicon Graphics *Indigo* workstation, which supports (stereo) sampled sound and MIDI I/O as standard features. It is, however, not as easily extensible as the SPARCstation-based system. Several other candidates fail because of having significantly less available CPU power (e.g., IBM or Apple PCs or NeXT machines), or because of the lack of off-the-shelf sampled sound and MIDI interfaces suitable to music (e.g., DECstations or the IBM RS/6000 machines). The key feature of the IDP is, however, the software environment.

Fig. 2. MODE system software components.



MODE: The Musical Object Development Environment

The MODE software system consists of Smalltalk-80 classes that address five areas: the representation of musical parameters, sampled sounds, events and event lists; the description of middle-level musical structures; real-time MIDI, sound I/O, and DSP scheduling; a user interface framework and components for building signal, event, and structure processing applications; and several built-in end-user applications.

Figure 2 is an attempted schematic presentation of the relationships between the class categories and hierarchies that make up the MODE environment. The items in Fig. 2 are the MODE packages, each of which consists of a collection or hierarchy of Smalltalk-80 classes. The items that are displayed in `courier` font are written in C and are the interfaces to the external environment. The paragraphs below introduce the packages in the object-oriented style—in terms of the state and behavior of hierarchical trees of related object types.

Using inheritance among Smalltalk-80 classes—the facility to specify software modules (classes) as specialized versions of other modules (their superclasses)—one first defines very general (abstract) classes for the tool kit, then more specific (concrete) subclasses of these that have more specialized messages for their particular behavior. The basic description of the packages, (e.g., *Magnitudes, Events/EventLists, Voices, or Sounds*) is the behavior of the abstract classes in each of these categories.

The SmOKe Music Representation

As displayed in Fig. 2, the classes related to representing the basic musical magnitudes (such as pitch, loudness, duration), and to creating and manipulating event and event-list objects form the "kernel" of the MODE. This package is known as the

Smallmusic Object Kernel (SmOKe—name contributed by Craig Latta and Danny Oppenheim).

SmOKe is a language-independent music representation, description language, and interchange format that was developed by a group of researchers during 1990 and 1991. The basic design requirements (SmOKe 1992) are that the representation support the following:

- abstract models of the basic musical quantities (scalar magnitudes such as pitch, loudness, or duration);
- instrument/note (voice/event, performer/music) abstractions;
- sound functions, granular description, or other (non-note-oriented) description abstractions;
- flexible grain-size of "events" in terms of "notes," "grains," "elements," or "textures";
- event, control, and sampled sound processing description levels;
- nested/hierarchical event-tree structures for flexible description of "parts," "tracks," or other parallel or sequential organizations;
- separation of "data" from "interpretation" (what vs. how in terms of providing for interpretation objects);
- abstractions for the description of "middle-level" musical structures, such as chords, clusters, or trills (may be optional);
- annotation and marking of event tree structures supporting the creation of heterarchies (lattices) and hypermedia networks;
- annotation including common-practice notation possible;
- description of sampled sound synthesis and processing models such as soundfile mixing or DSP;
- possibility of building converters for many common formats, such as MIDI data, Adagio, note lists, DSP code, instrument definitions, mixing scripts, or Pla (this is an application issue); and
- possibility of parsing live performance into some rendition in the representation, and of interpreting it (in some rendition) in real time (this is an application issue).

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 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 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.

MusicMagnitudes

MusicMagnitude objects are characterized by their identity, class, species, and value. Their 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, or *how* its value is stored. The mixed-mode music magnitude arithmetic is defined using the technique of *double dispatching* that allows message-passing based on more than one argument (i.e., the receiver of the message). These two methods—membership/species differentiation, and double dispatching—provide capabilities similar to those of systems that use the techniques of multiple inherit-

ance 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., `(440.0 Hz + 7 half-steps)` or `(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 MODE *Function* class hierarchy includes line segment, exponential segment, spline segment, and Fourier summation functions.

Events and EventLists

The Event object in the MODE 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 the 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). Note 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.

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 *Event* themselves. This means that event lists can behave like events and can therefore be arbitrarily deeply nested; that is, 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 mess ages 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.

Figure 3 shows the state of a typical MODE event list as an entity-relationship diagram where the arrows mean *has-a*. One can see the global dictionary for persistent event lists pointing to the given list, which has its own properties (if any), and a sorted collection of associations between durations and events (or sublists). The square boxes and gray arrows represent *associations* between the given name (as in a dictionary key or property name) and the value it "points" to. Each event has its own property dictionary and voice (if desired).

Fig. 3. Example Event and EventList state.

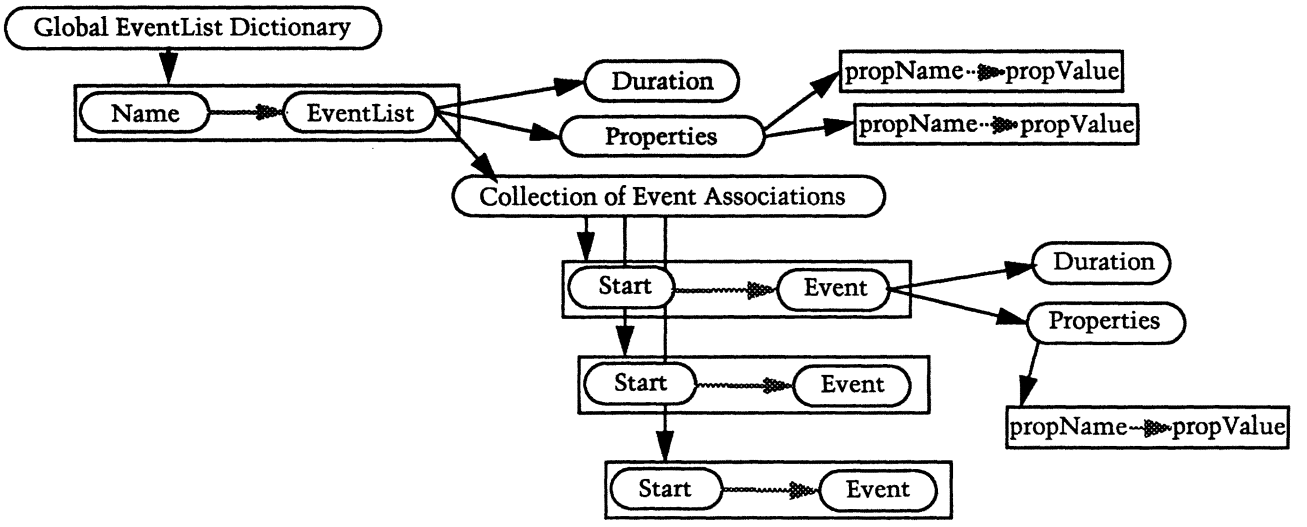


Fig. 3

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 time—it 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 1989). Several examples of the "verbose" format of SmOKe are shown in Fig. 4. The code examples use the Smalltalk-80 syntax whereby every message-

Fig. 4. Verbose SmOKe example.

```
"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 Amplitude 106."

"Event Creation Messages"
        "Create a `generic' event."
    Event duration: 1/4 pitch: `c3' ampl: `mf'
        "Create one with added properties."
    (Event dur: 1/4 pitch: `c3')
        color: #green; accent: #sfz

"EventList Usage"
        "Create a named event list."
    el := EventList newNamed: #demo1.
        "Add an event to it at time 0."
    el add: (Event dur: 1 pitch: 36 ampl: `mf');
        "Add an event after the first."
       add: (Event dur: 1 pitch: 40 ampl: `mf');
          "Add a sublist with 3 simultaneous
               events"
       add: (EventList new "a chord."
       add: (Event dur: 1 pitch: 36) at: 0;
       add: (Event dur: 1 pitch: 40) at: 0;
       add: (Event dur: 1 pitch: 43) at: 0)
```

Fig. 4

Fig. 5. Terse SmOKe Examples.

```
"Terse MusicMagnitude Creation
          using post-operators"
     440 Hz     250 msec
     1/4 beat   'c#3' pitch

"Terse Event creation using
          concatenation of music magnitudes"
     440 Hz, 1/4 beat, 44 dB.
     490 Hz, 1/7 beat, 56 dB,
               (#voice -> #flute).
     (#c4 pitch, 0.21 sec, 64 velocity)
               voice: Voice default.

"Terse EventList creation using
          concatenation of events or
          (duration event) asociations"

     (440 Hz, (1/1 beat), 44.7 dB),
     (1 => ((1.396 sec, 0.714 ampl)
          phoneme: #xu))

"Bach Example—First measure of Fugue 2
          from the Well-Tempered Klavier
          (ignoring the initial rest)."

     (((0 beat) => (1/16 beat,  'c3' pitch)),
       ((1/16 beat) => (1/16 beat, 'b2' pitch)),
       ((1/8 beat) => (1/8 beat, 'c3' pitch))),
     ((1/4 beat) => (1/8 beat, 'g2' pitch)),
     ((3/8 beat) => (1/8 beat, 'a-flat2' pitch)),
     ((1/2 beat) => ((1/16 beat, 'c3' pitch)),
       ((1/16 beat) => (1/16 beat, 'b2' pitch)),
     ((1/8 beat) => (1/8 beat, 'c3' pitch))),
     ((3/4 beat) => (1/8 beat, 'd3' pitch)),
     ((7/8 beat) => (1/8 beat, 'g2' pitch))
```

passing expression (and subexpression) starts with the receiver object and is followed by one or more message keywords and optional arguments. Class names in Smalltalk are always capitalized, while variable names are written lowercase. Comments are enclosed in double quotation marks ("...") in Smalltalk. In the verbose SmOKe format music magnitudes, events, and event lists are created by Smalltalk-80 instance creation messages sent to the appropriate classes. The first three expressions create various music magnitudes and coerce them into other representations. The second group shows the creation of two event objects with different properties. The last section creates a hierarchical event list with two note events followed by a sublist consisting of a chord. Figure 5 shows the most terse (and most common) form of SmOKe description; here again the first group of expressions describes several music magnitudes (this time using post-operators sent as messages to number or string objects), the second group creates note events, and the third shows a hierarchical event list. Note the use of 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)` means to build an event list with the two events as components. (A more detailed description of the SmOKe music representation can be found in SmOKe 1992.)

Persistency, Links and Hypermedia

The event and event-list classes have two special features that are used heavily in the MODE applications: *persistency* and *links*. Any MODE object can be made persistent by registering it in a shared dictionary under a symbolic name. Browsers for these (possibly hierarchical) dictionaries of (for example), persistent event lists, sounds, compositional structures, functions, or mixes are important MODE tools.

MODE objects also have behaviors for managing several special types of links, seen simply as properties where the property name is a symbol such as *usedToBe*, *isTonalAnswerTo*, or *obeysRubato*, and the property value is another MODE object, such as an event list. With this facility, one can build multimedia hypermedia navigators for arbitrary SmOKe networks. The three example link names 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.

EventGenerators and EventModifiers

The EventGenerator and EventModifier packages provide for music description and performance using generic or composition-specific middle-level objects.

Event generators are used to represent the common structures of the musical vocabulary such as chords, clusters, progressions, ostinati, or algorithms. Each event generator subclass knows how it is described (e.g., a chord with a root and an inversion, or an ostinato with an event list and repeat rate) and can perform itself once or repeatedly, acting like a Smalltalk-80 control structure. EventModifier objects generally hold onto a function and a property name; they can be told to apply their functions to the named property of an event list lazily or eagerly. (Event generators and modifiers are described in more detail in Pope 1991a.)

TR-Trees and Other Structures

The last components of the MODE music representation system implement higher-level description languages, user interface components, and/or compositional methodologies. For example, the *TR-Trees* system (Pope 1991b) is a rudimentary implementation of some of the ideas in Fred Lerdahl's *Generative Theory of Tonal Music* (Lerdahl and Jackendoff 1983).

MODE Event I/O and Performance

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.

Sampled Sound Processing and Support Classes

The objects that support sampled sound synthesis, recording, processing, and playback are grouped into several packages that support applications in several of the current synthesis/DSP paradigms, including Music-N-style synthesis, graphical interactive sample editing and arithmetic, tape recorders, MacMix-like mixers, and spatial localizers. The basic signal processing language is modeled after *mshell* and presents the model of a pocket calculator with mixed mode arithmetic and graphical inspector/editors on scalar, function, and (8-, 16-, 24-, or 32-bit) sample array data types. There are interfaces to higher-level analysis/synthesis packages in the form of Smalltalk user primitives (also known as foreign function calls) to C-language routines from Dick Moore and Paul Lansky that implement both phase and linear prediction-based vocoders, as well as sound I/O interfaces to both the 8- and 16-bit audio worlds. The standard soundfile formats (e.g., SPARCstation audio, NeXT, IRCAM) are supported for reading and writing different sample formats.

MODE System Interface

The interface between the MODE software and external formats and utilities is an important component of the system. In earlier versions of the MODE system (known as the HyperScore ToolKit), Smalltalk-80 user primitives were used to connect between the Smalltalk and C worlds. User primitives are Smalltalk-80 methods that call external C functions through interface functions that are linked into the Smalltalk virtual machine, similar to the way foreign function calls are used in Lisp. In the interest of portability, ease of implementation, and simplicity, this interface has been replaced by one that uses UNIX processes to communicate with external programs, "forking" them from within Smalltalk and either writing to their standard inputs or reading from their standard outputs. This kind of interface is used, for example, to play sounds, by having a special version of the *play* program that reads sample data from its standard input stream, or to read sound envelopes using the CARL *envelope* program and reading its

output stream. Another facility is the use of the CARL function generator routines (the GEN commands for cmusic), for creating Function objects by forking the GEN programs and reading their output data. More complex UNIX process interfaces are used for multistep processes such as linear prediction (LPC) analysis, where one Smalltalk method calls the cmix lpc analysis program, the pitch tracker, and the merge program that combines lpc and pitch data into one file, which is then read into Smalltalk. While this interface is much simpler than one using user primitives, it makes the system less easily portable to non-UNIX platforms, and requires the presence of components of both the Princeton cmix and U. C. San Diego CARL software distributions. Another kind of interface is via external file formats that are known to MODE classes. Examples of this type of interface are sound files, where the SoundFile class can read and write NeXT/SPARC, IRCAM or cmix sound file headers, or the special voice classes that can generate note list files for the cmusic, cmix, or csound software sound synthesis packages.

MODE User Interface Components

One of the most powerful aspects of the Smalltalk-80 programming system is the interactive user interface framework, called MVC for *Model/View/Controller* programming (Krasner and Pope 1988). Using MVC means that the user first creates a *model* object with domain-specific state and behavior (e.g., an event list or a sampled sound). Interactive applications that manipulate instances of this model consist of two objects, the *view* and the *controller*, which can be thought of as output and input channels, respectively. Views generate a textual or graphical presentation of some aspect(s) of the state of the model on the display, and controllers read the input devices and send messages to the model or the view. The advantage of this design is that many applications can use the same set of view and controller components, assuming a good library of them is available.

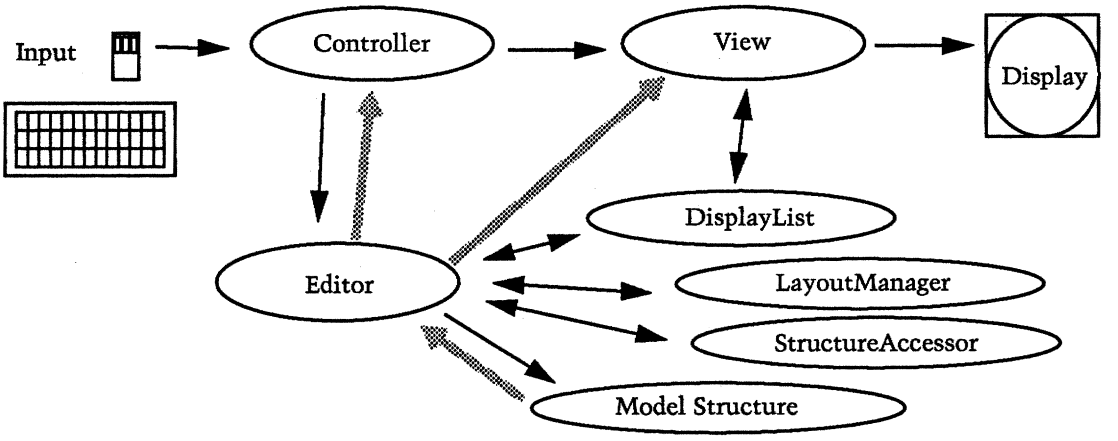
The Smalltalk-80 development tools and built-in applications reuse a small and well-implemented set of view and controller classes to present the user with various windows that are seen as *inspectors*, *editors*, or *browsers*. An inspector allows one to "see inside" an object and to manipulate its static data interactively by sending it messages. An editor provides a higher-level, possibly graphical, user interface to an object. Browsers organize groups or hierarchies of objects, possibly of the same or similar types, and allow one to select, interact with, and organize these objects into some sort of hierarchy. The MODE user interface is organized as inspectors, editors, or browsers for various families of musical parameters, events, signals, and structures, examples of which are presented below.

Navigator MVC Framework

The version of MVC used in the MODE is based on the *Navigator* framework, an extended MVC support library developed at ParcPlace Systems, Inc., by David Leibs, Adele Goldberg, and the author (Pope, Harter, and Pier 1989). Figure 6 shows the architecture of a Navigator MVC application. The model and the controller are normally relatively simple objects. The view holds onto a display list (i.e., a structured graphics representation) that it displays in its visual field. The editor object knows about the model and also the current selection and other editor-specific information. It responds to messages from the view about layout manager selection and to messages from the controller about user actions. Editors can also regenerate the view's display list using the layout manager (to generate the list) and the structure accessor (to get at the model's properties). The gray arrows in Fig. 6 denote indirect message-passing via Smalltalk-80's *dependency mechanism*. This means that the editor, for example, does not explicitly know the identities of the view and controller, but rather broadcasts messages about its state changes, which the dependency mechanism forwards to the (zero, one or more) dependent objects. The same relationship applies between the editor and the underlying model data structure object.

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/subnodes?

Fig. 6. Navigator Model/ View/Controller software architecture.



Who are they? Add this child to them). One specific, concrete subclass might know how to apply that language to navigate through a hierarchical event list (by querying the event list's hierarchy); another concrete structure accessor might use the same messages to traverse a hierarchical document (as in an outliner). The role of the *LayoutManager* object is central to building Navigator MVC applications. The MODE'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 application objects in the system.

Much of the work of building a new tool within the MODE 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, or "MacDraw with structures behind it"), and that the bulk of many applications' special functionality resides in a small number of changes to existing accessor and layout manager classes.

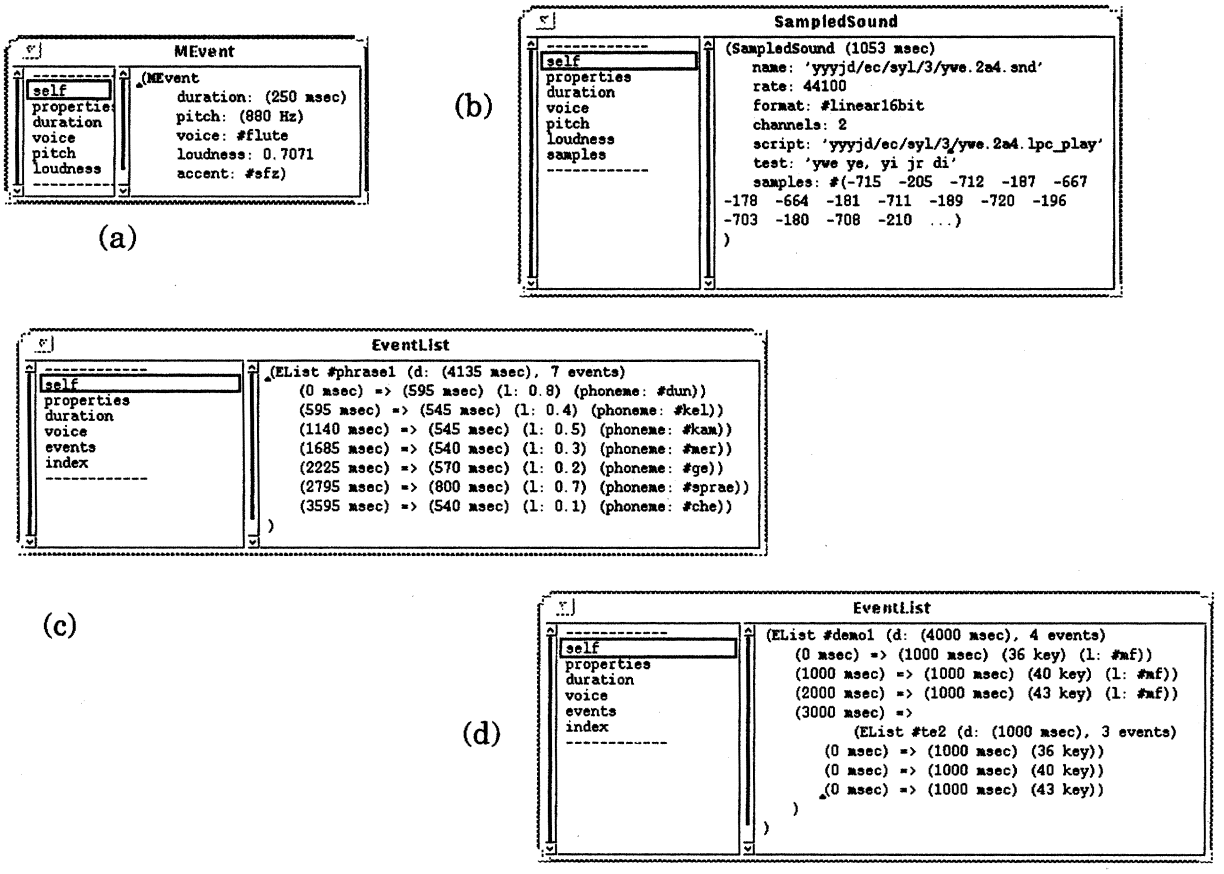
MODE Applications in IDP: Examples

This section presents several examples of the current crop of IDP/MODE graphical user interfaces. Several other IDP tools are shown in the image on the cover of this issue of *Computer Music Journal* and described on page 2.

Event and EventList Inspectors and Browsers

Figure 7 illustrates the user interface of simple inspectors on MODE events and event lists. The left-hand pane of the inspectors shows a list of the instance variables or components of the objects being inspected. The right-hand pane displays a pretty-printed rendition of the selected item. Figure 7 shows an event inspector looking at a simple event (a), a sampled sound object inspector (b), a simple event list in an inspector (c), and a two-level event list in which the fourth event is a sublist (d).

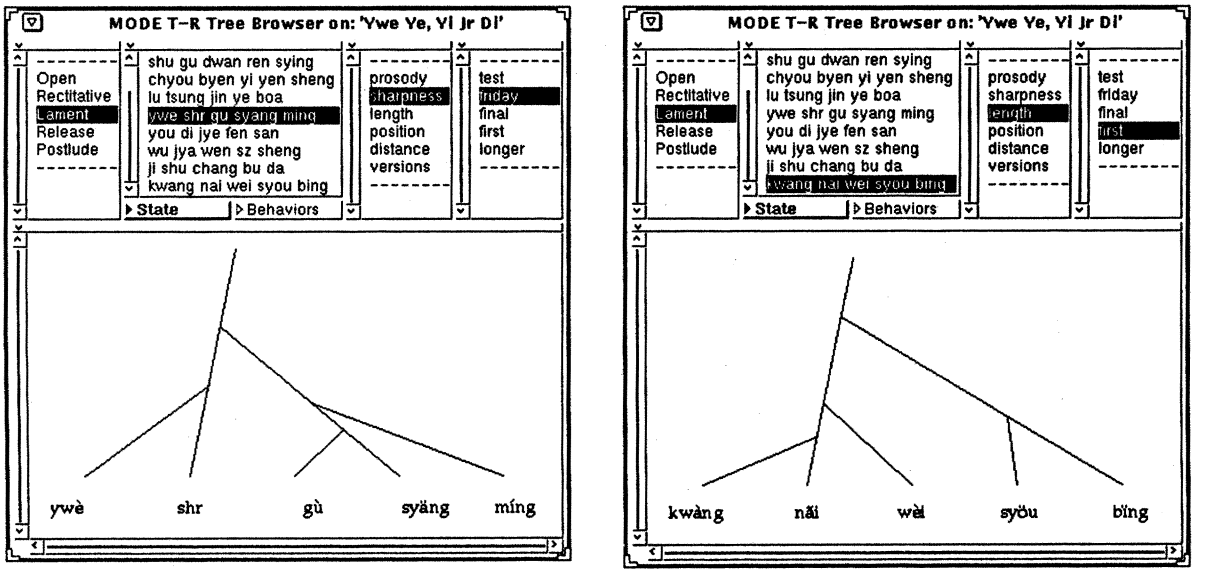
Fig. 7. MODE Event, Sound, and EventList inspectors.



EventList Editors and Music Notation

Score or music notation editors are called event list editors in the MODE; an example is shown in the view on the lower-right of the cover image of this issue of *Computer Music Journal*, which shows an editor using Hauer-Steffens notation (developed by Josef Matthias Hauer and extended by Walter Steffens for the notation of 12-tone music). In this particular editor, the note heads are used to denote the voice upon which an event is to be played. The event's duration is shown by the blue horizontal line extending from the note head to the right. The amplitude of the note is shown by the red line extending up from the note head; its position (in a stereo space) is displayed by the green line extending up or down from the note head. The note's envelope is shown in red, and the yellow line under the duration signifies the event's accent. This is just one example of an event list editor that implements an alternative notation; it can easily be extended by customizing the layout manager object to position the events according to different rules, or to generate different types of event displays. One could also implement common-practice Western music notation (CPN), though this is a rather difficult task due to the complexity of CPN; it has not been implemented in the MODE to date primarily due to lack of interest.

Fig.8. MODE TR-Tree editor examples.



TR-Tree Editors and Browsers

Two examples of one type of TR Tree browser view are shown in Fig. 8 (from Pope 1991b). The texts are two sentences from the poem *Moonlight Night (Ywe Ye, Yi Jr Di)* by the T'ang dynasty Chinese poet Du Fu. The four list views along the top of the browser show *sections, phrases, theories,* and *trees.* The sections shown are the second movement of *Celebration;* within the selected section (*Lament*), one can see the eight lines of the poem. For each line, we have a collection of tree types (theories), as shown in the third list; and there can be many trees within each theory—versions of one TR-Tree weighting.

For the left-hand example, you are looking at the "sharpness" tree of the "friday" version of the fourth line of the poem ("The moon is much brighter over my hometown tonight"); this shows a strong dominance of "shr" and "syang" syllables. According to way the "sharpness" theory is implemented, changing the weights of this tree (by "dragging" a node in the tree with the mouse) would change the filter coefficients for the synthesis of the relevant syllables (thereby changing their "sharpness"). The right-hand view shows one "length" tree of the last sentence of the poem ("And the on-going war only makes matters worse"), where the "nai" is being exaggerated in length. Editing this tree would change the relative durations of the syllables.

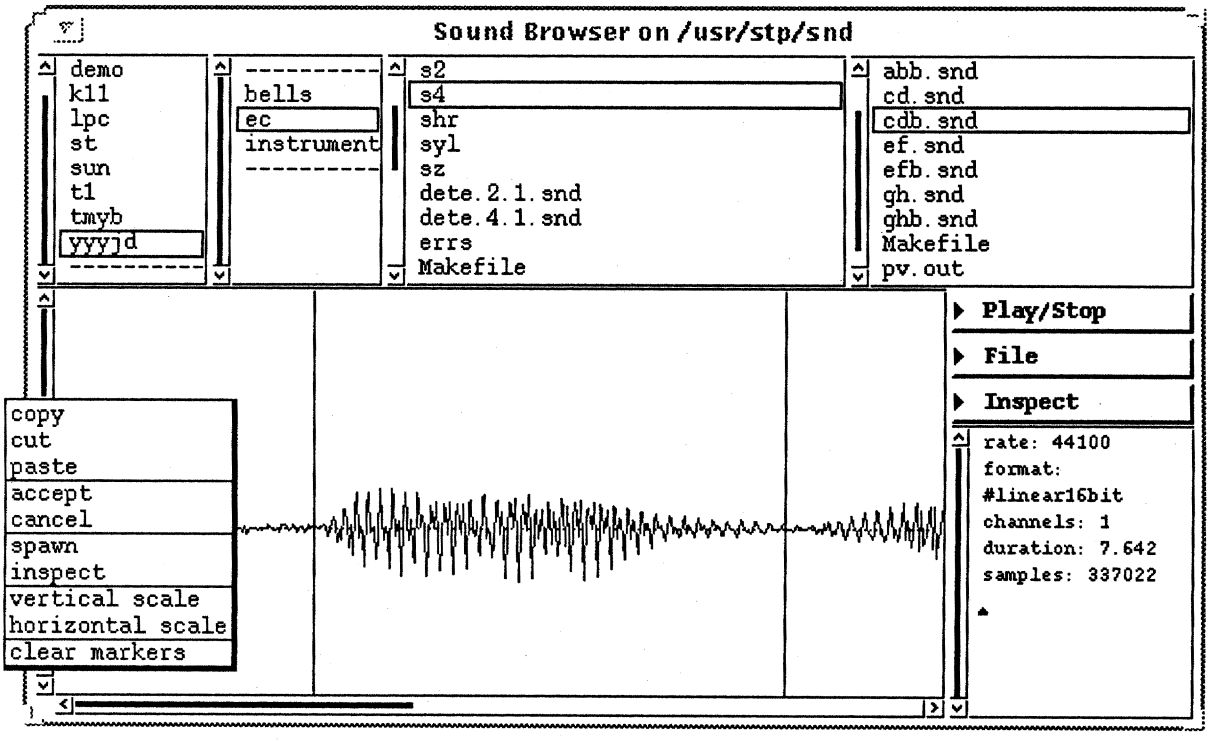
Sound Processing, Editing, and Mixing

The primary sampled sound interface is the sound browser view, shown in Fig. 9. The lower portion of the browser view is a sound editor view; it supports the basic sampled sound operations shown in the popup menu visible in the view. The sound browser allows users to manipulate a hierarchy of sound dictionaries, such as the four-level tree shown in the upper list views of the view.

Other Interactive Tools

Three other tools are displayed in the cover image that deserve mention here: LPC frame data editors, mix views and the play list view. The interface to linear prediction-based analysis and cross-synthesis uses Paul Lansky's cmix utilities; the problem is that one often needs to "massage" LPC analysis data to "smooth out" the pitch or residual functions. The LPC frame data edi-

Fig. 9. MODE Sound browser.



tor allows one to view LPC analysis data, and "clean it up" by interpolating data between two chosen points (shown in the cover image by the white cross-hair icons on the two selected points on the pitch function).

The mix view is a cheap "work-alike" of Adrian Freed's MacMix program (which is used as the user interface to the Studer Editech Dyaxis digital mixer). In this view, one can load any number of soundfiles, orient them in time, apply envelopes to them, position them in a multichannel space, and perform mixing. The current version of this tool uses cmusic as the actual mixer, writing mix objects (a special kind of event list) to files as cmusic note lists. This means that there is more overhead involved in mix performance, but simplifies the implementation immensely. A future version of the MODE would implement mixing directly using Smalltalk user primitives so that simple mixes can be performed in real time.

The play list (shown in the center-right of the cover image), is a relative of the sound browser without the fancy graphical display. It is very useful for sound file management and can shows lists of the contents of sound file directories and display the header data of sound files that are selected from the list. One can play sound files from within the play list, and can write comments into them—necessary for file management in large compositions.

Conclusions and Directions

There are three directions of current and future development of the IDP hardware and software: extension of the tool set; ports of the system to ever more powerful host platforms; and extension of the external interfaces to support more I/O media and channels. Most of the current work in the tool arena is centered around better support for compositional structures (e.g., TR-Trees and Petri nets used for deterministic composition) and extensions to the set of event list editors, so that new notations are available

to support specific types of events or structures. The collection of event generator classes is also being extended.

Moving the IDP/MODE system to a new platform is relatively easy if two conditions are met—the platform must use the UNIX operating system (or some reasonable variant thereof), and ParcPlace Systems Objectworks\Smalltalk version 4.1 or newer must be available on the platform. By the time this article appears in print, both of these conditions are expected to be met by the Silicon Graphics Inc. Indigo workstation and the third-generation NeXT machine. Several aspects of the system would be simplified by moving it to either of these platforms, since both of them include 16-bit DACs and DSP coprocessors as "standard equipment."

It is also hoped that an interface (via MIDI) to the STEIM SensorLab analog input processor can be developed, this being held up at present by the lack of an available SensorLab. This interface would allow performance with the IDP using alternative input devices such as "Hands-style" controllers (work in progress).

A third possible direction for future development is the porting of the MODE software to other Smalltalk dialects, such as the (free) GNU Smalltalk. The two problems at present are the lack of higher-level software tools in GNU Smalltalk, and its relatively poor performance (relative to Objectworks\Smalltalk). Both of these factors are, however, functions of time.

The *Interim DynaPiano* or IDP system is the newest in a series of workstation-based composer's tools and instruments. The system is based on a powerful modern UNIX workstation and a flexible object-oriented software environment. The system costs about $20,000 in its present configuration and is transportable. The bulk of the MODE software described here is available free for noncommercial distribution via anonymous network *ftp* and can be found in the directory "pub/st80" on the InterNet machine named "ccrma-ftp.stanford.edu." There are also PostScript and ASCII text files of several other MODE-related documents there, including an extended version of this article.

The prime advantages of the system are its power and flexibility, and the extensibility and portability of the MODE software. The main problems are the (still relatively high) cost of the system's hardware components, and the complexity of the software. The first of these is a function of time, and the current price-performance war among the engineering workstation vendors can be expected to deliver ever more powerful and cheaper machines in the future.

At the time of this writing (Spring, 1992), the system described here is basically functional and in use in realizing the second and third movements of *Celebration.* The components that are still in-progress are the mixing views, the Music-N and localizer classes, and the primitive interfaces to the external vocoder functions (I still use the shell for this).

This article has presented the current IDP system as a member of a family of systems. I believe we can expect the profile of the core technology of IDP-like systems to remain stable for a number of years to come.

Acknowledgments

It would be extremely unfair to present the design or the implementation of the MODE as the work of the author alone. Over several years and versions of the HyperScore ToolKit, a number of users and programmers contributed design input, class implementations, and user interfaces to the system. Among those who were most present during the process are certainly Mark Lentczner (whose scheduler is still the simplest Smalltalk-based one in use), Lee Boynton and Guy Garnett (who contributed the primitive scheduler used for MIDI performance), John Maloney (who sent in extensions to many parts of the system), Glen Diener and Danny Oppenheim (whose applications showed many of the weaknesses of the system), and Hitoshi Katta and John Tangney (both of whom contributed new implementations of important system components). Designs and ideas for parts of the system were also taken from the writings of, and numerous delightful discussions with, Roger Dannenberg, Lounette Dyer, D. Gareth Loy, and Bill Schottstaedt. The SmOKe representation was developed and refined by a group that included Danny Oppenheim, Craig Latta, Guy Garnett, and Jeff Gomsi, whom I thank for their input.

References

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

Buxton, W., et al.1978. "An Introduction to the SSSP Digital Synthesizer." *Computer Music Journal* 2(4): 28–38. Reprinted in C. Roads and J. Strawn, eds. 1985. *Foundations of Computer Music*. Cambridge, Massachusetts: MIT Press.

Buxton, W., et al.1979. "The Evolution of the SSSP Score Editing Tools." *Computer Music Journal* 3(4): 14–25. Reprinted in C. Roads and J. Strawn, eds. 1985. *Foundations of Computer Music*. Cambridge, Massachusetts: MIT Press.

Cadmus. 1985. "Cadmus Computer Music System Product Announcement." *Computer Music Journal* 9(1): 76–77.

Computer Audio Research Laboratory. 1983. *CARL Software Startup Kit*. La Jolla, California: CARL, University of California in San Diego.

Dannenberg, R. B. 1989. "The Canon Score Language." *Computer Music Journal* 13(1):47:56.

Deutsch, L. P., and E. A. Taft. 1980. "Requirements for an Experimental Programming Environment." Research Report CSL-80-10. Palo Alto, California: Xerox PARC.

Fry, C. 1984. "Flavors Band: A Language for Specifying Musical Style." *Computer Music Journal* 8(4):20–35. reprinted in S. T. Pope, ed. 1991. *The Well-Tempered Object: Musical Applications of Object-Oriented Software Technology*. Cambridge, Massachusetts: MIT Press.

Goldberg, A., and S. T. Pope. 1989. "Object-Oriented Programming Is Not Enough!" *American Programmer: Edward Yourdon's Software Journal* 2(7):46–59.

Krasner, G., and S. T. Pope. 1988. "A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80." *Journal of Object-Oriented Programming* 1(3):26–49.

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

Learning Research Group (LRG). 1976. *Personal Dynamic Media*. Research Report SSL-76-1. Palo Alto, California: Xerox PARC

Lerdahl, F., and R. Jackendoff. 1983. *A Generative Theory of Tonal Music*. Cambridge, Massachusetts: MIT Press.

Lindemann, E., et al. 1991. "The Architecture of the IRCAM Musical Workstation." *Computer Music Journal* 15(3):41–49.

Pope, S. T. 1982. "An Introduction to *msh*: The Music Shell." In *Proceedings of the International Computer Music Conference*. San Francisco: International Computer Music Association.

Pope, S. T. 1991a. "Introduction to the MODE." In S. T. Pope, ed. *The Well-Tempered Object: Musical Applications of Object-Oriented Software Technology*. Cambridge, Massachusetts: MIT Press.

Pope, S. T. 1991b. "A Tool for Manipulating Expressive and Structural Hierarchies in Music (or, `TR-Trees in the MODE: A Tree Editor Based Loosely on Fred's Theory')." In *Proceedings of the International Computer Music Conference*. San Francisco: International Computer Music Association.

Pope, S. T., N. Harter, and K. Pier. 1989. *A Navigator for UNIX*. Video presented at the 1989 ACM SIGCHI Conference. Available from the Association for Computing Machinery (ACM), New York.

Rodet, X., and P. Cointe. 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, Massachusetts: MIT Press.

Scaletti, C. 1989. "The Kyma/Platypus Computer Music Workstation." *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, Massachusetts: MIT Press.

SmOKe (Smallmusic Object Kernel Group). 1992. "The Smallmusic Object Kernel: A Music Representation, Description Language, and Interchange Format." Document named "OOMR.ps.Z" available via Internet file transfer from the server "ccrma-ftp.stanford.edu" in the directory "pub/st80."