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

Stephen Travis Pope
The Nomad Group, *Computer Music Journal*, and
CCRMA: Center for Computer Research in Music and Acoustics
Department of Music, Stanford University
P. O. Box 60632, Palo Alto, California USA
Electronic mail: stp@CCRMA.Stanford.edu

D R A F T

Abstract

The *Interim DynaPiano* (IDP) is a tool and instrument for music composition and performance based on a UNIX workstation computer and object-oriented Smalltalk-80 software components. The IDP hardware consists of a powerful RISC CPU with large RAM and disk memories, a hardware-accelerated color graphics system, and interfaces for real-time sampled sound and MIDI I/O. The Musical Object Development Environment (*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 by a professional composer (i.e., around the price of a good piano or MIDI studio). 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 paper presents an overview hardware and software components of the current IDP system.

Introduction

The *Interim DynaPiano* (IDP) is an integrated computer hardware/software configuration for music composition, production, and performance based on a Sun Microsystems Inc. SPARCstation-2™ computer and the Musical Object Development Environment (*MODE*) software. The SPARCstation is a powerful RISC- (reduced instruction set computer) based workstation computer running the UNIX™ operating system. The MODE is a large family hierarchy of object-oriented software components for music representation, composition, performance, and sound processing; it is written in the Objectworks\Smalltalk-80™ language and programming system.

This paper presents an overview hardware and software components of the current IDP system. The background section below 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 paper is a description of the MODE signal and event representations, software libraries, and application examples.

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 1970's was *Personal Dynamic Media* (LRG 1976), a report that described the Smalltalk-76 system developed by Alan Kay, Adele Goldberg, Dan Ingalls, Larry Tesler, Ted Kaehler, L. Peter Deutsch Dave Robson, et al. 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 computers. Because the name "Alto" was a Xerox-internal code name, the authors were not allowed to use it, so they presented the system they and their colleagues had developed at PARC as an *Interim DynaBook* machine (the Alto PC), with the "interim" software environment Smalltalk-76. There are several sound and music applications included in the examples they present. In the spirit that each machine and each software system is “interim,” I 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 programming environment and has adequate real-time signal *and* event processing capabilities to be used both as a stand-alone 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 based on 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 re-implementation of the HyperScore ToolKit undertaken in 1990-91, is based on the representations and tools used in the recent realization entitled *Kombination XI* (1990) (Part 1 of *Celebration*, work in progress). 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 MODE and the IDP work 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 vs. power and flexibility; and real-time vs. power. Much modern software is designed to be extremely easy to learn, to the detriment of its power and flexibility. The Macintosh vs. UNIX debate is a prime example of this—the Macintosh can be seen as a computer for children, and a UNIX machine as one for “consenting adults.” IDP and the MODE software in particular, is a large and complex system which is non-trivial 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 doable 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.

Background

The development of interactive workstation-based computer music systems started very shortly after personal computers with enough power and flexibility to support such applications became available. By the mid-1970’s, 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).

Integrated Workstation-based Computer Music Systems

The design and configuration of the IDP system described here is the result of 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 ICMC in Paris). These components are:

- commercial engineering workstation (e.g., 680X0- or RISC-based) computer;
- large RAM and soundfile disk memories;
- multi-tasking 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/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 multi-tasking 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-turn-around 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 meats the requirements of *exploratory programming systems* as defined in (Deutsch and Taft 1980) or *interactive programming environments* as defined in (Barstow, Shrobe, and Sandewall 1985). 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:

- a powerful, abstract programming language with automatic storage management, interpreted and/or incrementally-compiled execution, and a run-time available compiler;
- software libraries including reusable low- and high-level modules;
- a windowed interactive user interface "shell" text and/or menu system;
- a set of development, debugging, and code management tools;
- an interface to "foreign-language" (often C and assembler) function calls; and
- a 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 (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 (East coast, West coast and European), 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 is a sophisticated development environment that is also portable, fast, commercially supported, and stable. Other commercial (e.g., from Digitalk) or public domain (e.g., GNU) Smalltalk systems 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;`), and may not generally assign other kinds of data to a variable after its declaration (e.g., `i = "hello";`). 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 in a valuable language asset because of the increase it brings in 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` 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 multi-media applications.

IDP-like systems of the 1980's

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. I will comment on several of the systems that have influenced the current design before presenting IDP in detail. The research systems that appeared along the coasts of the USA in the mid-to-late 1970's were generally programmed in non-mainstream languages and used non-commercial (i.e., non-available) hardware. The advent of UNIX and the Motorola 68000 microprocessor lead to a plethora of commercial UNIX workstation computers in the first years of the 1980's that went by the moniker "3M machines" —1 MIPS CPU performance, 1 MB RAM memory and 1 MPixel display resolution. Several groups used these to develop *computer music workstations, intelligent composer's assistants* or early IDP-like systems.

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 et al. at the University of Toronto in the late 1970's. SSSP combined a DEC PDP-11 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 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 integration and use of the tools is 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, Apple Macintoshs, and NeXT machines.

The *Cadmus 9230/m* (for music) workstation configuration developed by Günther Gertheiss, Ursula Linder, and myself at PCS/Cadmus computers in Munich in 1983 and 1984 was based on an asymmetrical multiprocessor with 68010 CPUs for the operating system, the window system, the ethernet driver, the file system, and optional terminal or modem packetizing and multiplexing (Cadmus 1985). The DAC/ADC system was based on a (then new) Sony convertor evaluation card with a parallel QBus interface and buffer card. The system had 2-4 MB RAM and 220 MB of disk memory in the default configuration and cost $35,000. Two of these systems were sold, and one of them is still in operation.

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 notelist 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* is a Symbolics Lisp Machine-based tool kit based on 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 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 by Xavier Rodet et al. at IRCAM between 1980 and 1985 used (Obj)VLisp 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 interfaces, connection to the CHANT synthesis language, and the *Esquisse* composition system.

Kyma is a graphical sound manipulation and composition language developed by Carla Scaletti. It is linked to the *Capybara*, a scalable real-time digital signal processor built by Kurt Hebel. The Smalltalk-80-based software tools that comprise Kyma 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 multi-language integration with Smalltalk-80 methods generating, downloading, and scheduling micro-code 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 CCRMA Center for Computer Research in Music and Acoustics at Stanford University combines a NeXT "cube" workstation with an Ariel Corp. *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.

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

There are several non-examples I would like to site 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 the event and event list levels only (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), that are not programmable "composer's workbenches." Such applications must be provided in an open and customizable way for an IDP to be "dyna."

Smalltalk-80 Software for Music

In the later 1980's, a number of object-oriented frameworks and tool kits for event and signal processing for music (in addition to those already mentioned), were developed and reported in various languages. The Smalltalk-80 literature includes *MusicCategory* (McCall), *SoundKit* (Lentczner), *T-Trees* (Diener), *VDSP* (Mellinger, Garnett, and Mont-Reynaud), the *NoteEvent* system (Maloney), *TRAX* (Toenne), *AMUSED* (Böcker and Mahling), and *Javelina* (Hebel) (to name quite a few).

The higher-levels of the IDP software are primarily implemented as a set of packages managed as Smalltalk-80 class hierarchies that have evolved (more-or-less) continuously since 1984. The basic class structures and behaviors (and therefore the representation), were modeled after the (1983) ARA system in Lisp, and has gone under the names *SmallSong*, *DoubleTalk*, *HyperScore ToolKit*, *Topaz*, and now *MODE*. We will describe the MODE in more detail below.

IDP Hardware Configuration

Figure 1 shows the configuration of the current mobile IDP system. The general-purpose and audio/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 MB RAM, 64 kB cache memory;
  210 + 660 MB SCSI disk memory;
  streamer tape cartridge;
  color graphics accelerator;
  16” Sony color monitor, keyboard, mouse;
  internal CODEC (telephone-quality) DAC/ADC; and
  Telebit high-speed modem.
- Ariel Corp. *S56X* SBus card with:
  Motorola DSP56001DSP coprocessor;
  64 kB local RAM memory;
  SBus DMA controller for interface to SPARCstation;
  programmable Xilinx coprocessor; and a
  NeXT-compatible (DSP56000 SSI/SCI) DSP-Port connector.
- Ariel Corp. *ProPort* out-board DAC/ADC with
  pro-audio-quality 44.1 or 48 kHz stereo I/O; and
  balanced analog audio I/O connectors.

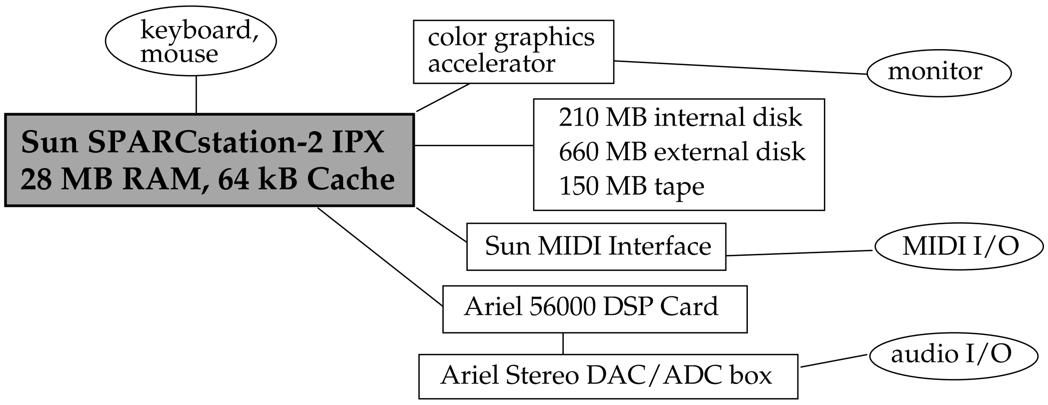


Figure 1: Current IDP hardware configuration

The main differences between this system and the older ones mentioned above is quantitative. The CPU power and memory capacities have increased roughly eight- to ten-fold relative for example to the Cadmus system of 8 years ago, and the dedicated DSP and scheduling facilities have been drastically improved. The hardware system is now small enough to travel (in two flight cases that are approximately 50 cm cubes), 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. The large RAM (28-40 MB) 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.

MODE: The Musical Object Development Environment

The MODE software system consists of Smalltalk-80 classes that address five areas: (1) the representation of musical parameters, sampled sounds, events and event lists; (2) description of middle-level musical structures; (3) real-time MIDI, sound I/O and DSP scheduling; (4) user interface framework and components for building signal, event, and structure processing applications; and (5) several built-in end-user applications. We will address each of these areas in the sections below.

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 Figure 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.

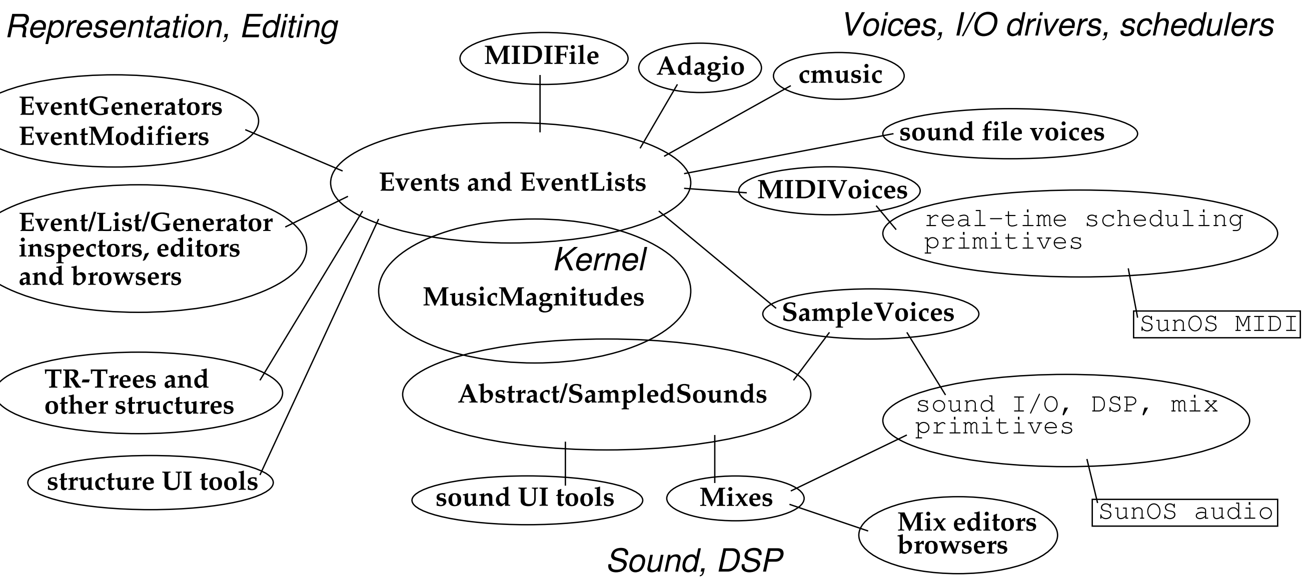


Figure 2: MODE system software components

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.

MODE Music Representation

As displayed in Figure 2, the “kernel” of the MODE are the classes related to representing the basic musical magnitudes (such as pitch, loudness, duration, etc.), and for creating and manipulating event and event list objects.

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 for “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 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 are defined, e.g., “`440.0 Hz + 7 half-steps`,” or “`1/4 Beat + 80 msec`.”

The basic model classes include *Pitch*, *Loudness*, *Duration*; exemplary extensions include *Length*, *Sharpness*, *Weight*, and *Breath* for composition or notation specific magnitudes. Figure 3 shows several examples of desirable notations for music magnitudes, along with their value classes and species. The actual class names of these objects (rarely seen by the average user) will be ugly things like *HertzPitch*, *SymbolicLoudness* or *MillisecondDuration*.

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, as in the examples shown in Figure 3, 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.

		Magnitude model (species)		
		Pitch	Amplitude	Duration
Implementation model (member)	Integer	36—key number	64—MIDI velocity	500—msec
	Float	440.0—frequency (Hz)	0.7071—ampl. ratio	1.0—sec.
	String	'a5'—note name	'mf'—dynamic name	'1/2'—'beats'
	Fraction	3/2—ratio to 'root'	1/3—ampl. ratio	3/4—'beats'

Figure 3: Examples of MusicMagnitude class membership and species

Events and EventLists

The *Event* object in the MODE is basically just a property-list dictionary with a duration. There are music-specific subclasses that have special knowledge of music magnitudes such as pitch, loudness, and voice. 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 notation editor. It is common to have events with complex objects as properties such as 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 different by various voices or structure accessors.

EventList objects hold onto collections of events that are tagged and sorted by their start times (represented as the durations between the start time of the event list and that of the event, thereby avoiding the question of time altogether). The event list classes are subclasses of *Event* themselves. This means that event lists can behave like events and can therefore be arbitrarily-deeply hierarchical, 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 (sorted by relative start time)—`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*, meaning 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 4 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 sub-lists). 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).

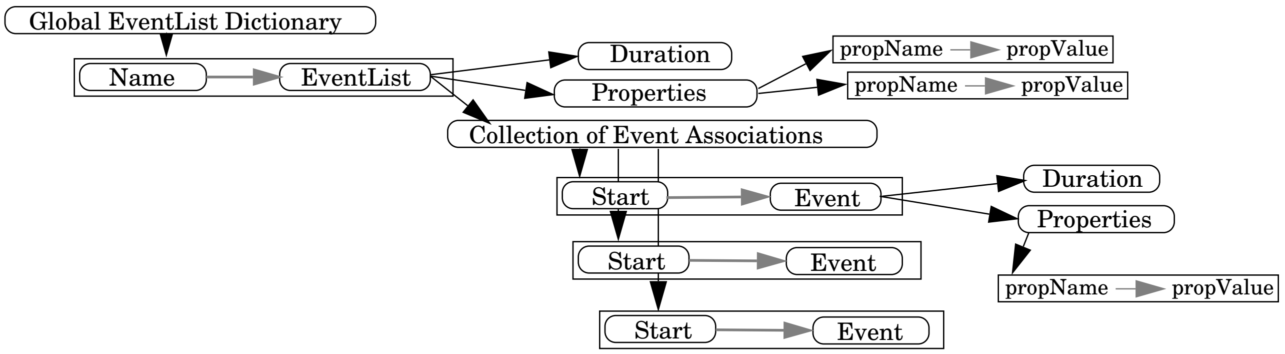


Figure 4: Example Event and EventList state

In Figure 5 you see the structure of a typical hierarchical score, where decomposition has been used to manage the large number of events, event generators and event modifiers necessary to describe a full performance. The arrows here mean *is-component-of*. The score is a tree (possibly a forest, i.e., with multiple roots) of hierarchical event lists representing sections, parts, tracks, phrases, chords, or whatever abstractions the user desires to define. The MODE does not define any fixed event list subclasses for these types—they are all various compositions of parallel or sequential event lists.

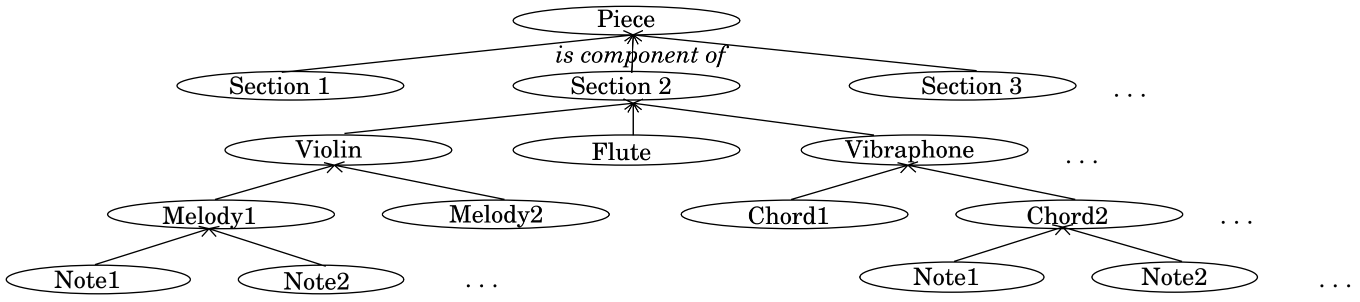


Figure 5: Example score as an event list hierarchy

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)). There is a small number of event subclasses such as NoteEvent and EventList, rather than a large number of them for different types of input or output media (such as in systems with MIDI, *cmusic*, and DSP event types). The fact that the MODE's simplest textual event and event list description format consists of 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).

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 (e.g.,) 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, e.g., an event list. With this facility, one can built multimedia hypermedia navigators for arbitrary network or web types and profiles. 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.

Figure 6 shows a collection of events and event lists that are related via several types of links. These are examples of the types that might be used in a hypermedia browser/navigator for score annotation, code and version management or multimedia linking between different categories of documents and schedules.

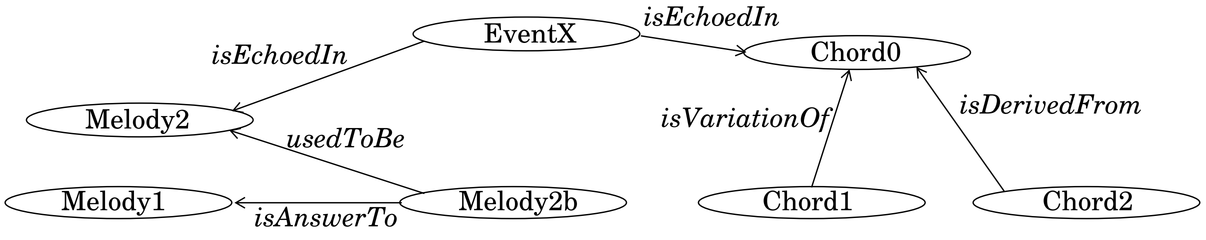


Figure 6: Examples of Link usage and types

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, looking 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 (batch or real-time mode) or eagerly (now). The examples at the end of this document present the usage of event generators and modifiers in more detail.

TR-Trees and Other Structures

The last components of the MODE music representation system are the packages the implement higher-level description languages, user interface components, and/or compositional methodologies. The *TR-Trees* system (Pope 1991c) is a rudimentary implementation of some of the ideas in Fred Lerdahl's *Generative Theory of Tonal Music*. The *DoubleTalk* system (Pope 1986) (currently being revived), is a group of interactive tools for editing and executing musical models described as logic-marked Petri nets. Both of these are described in length elsewhere.

MODE Event I/O and Performance

The "performance" of events takes place via *Voice* objects. Events have properties that are 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 or external description languages. There are voice "device drivers" for common file storage formats—such as cmusic notelists, the Adagio language, MIDIFile format, phase vocoder scripts—or for use with real-time schedulers and interfacing to MIDI or sampled sound drivers. Some voices also hold onto I/O port objects such as file streams or MIDI drivers; these classes can be refined to add new event and signal file formats or multi-level mapping 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 co-process), 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.

The *MIDIVoice* classes normally send messages to an object that is a direct interface to the Sun operating system's MIDI scheduler/driver. There is also a Smalltalk-80 version of a simple scheduler for portability, and compatible driver interfaces for other machines. MIDI voices talk to *MIDIPort* objects, so that it is even possible to have multiple MIDI I/O streams, or to use special messages for the specific devices attached to MIDI as mentioned above.

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 (AKA 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 sound file formats (e.g., SPARCstation audio, NeXT, IRCAM) are supported for reading and writing different sample formats.

MODE User Interface Components

One of the most powerful aspects of the Objectworks\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, provided a good library is available.

The Objectworks\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 myself (Pope, Harter, and Pier 1989). Figure 7 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., structured graphics representation) that it displays and scrolls/zooms in its visual field (a graphics context). 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 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 the figure 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.

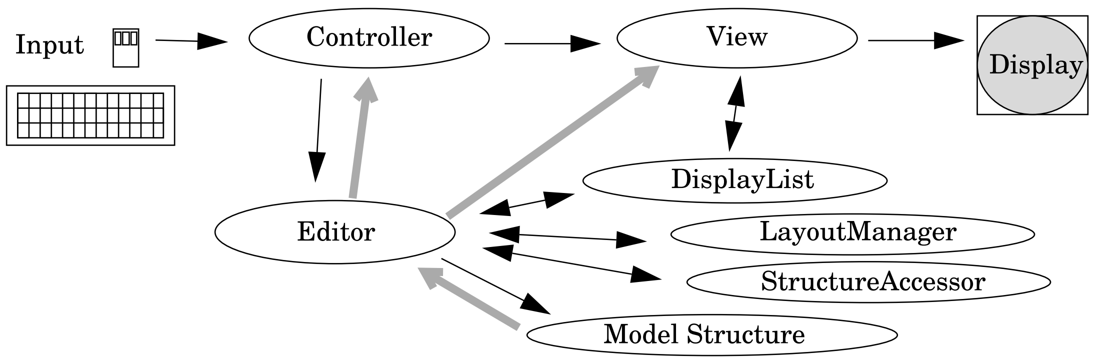


Figure 7: Navigator Model/View/Controller Software Architecture

A *StructureAccessor* is an object that acts as a translator or protocol convertor. An example might be an abstract one 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 some aspect of the event list's hierarchy); another concrete structure accessor might use the same messages to traverse a class inheritance hierarchy or 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 customization work of building a new event list editor within the MODE often goes into customizing the interaction and manipulation mechanisms, rather than just the layout of standard pluggable view components. Users can easily extend the range of interaction paradigms, building editors with powerful "accelerators" or macro capabilities, or mixed editor/inspector views. 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

The following sections present several examples of the current crop of IDP/MODE applications. The code examples use the Smalltalk-80 syntax whereby every message-passing expression (and sub-expression) 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 quotes ("...") in Smalltalk.

Event and EventList Description Formats

Figure 8 displays several description code examples of events and events lists in MODE.

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

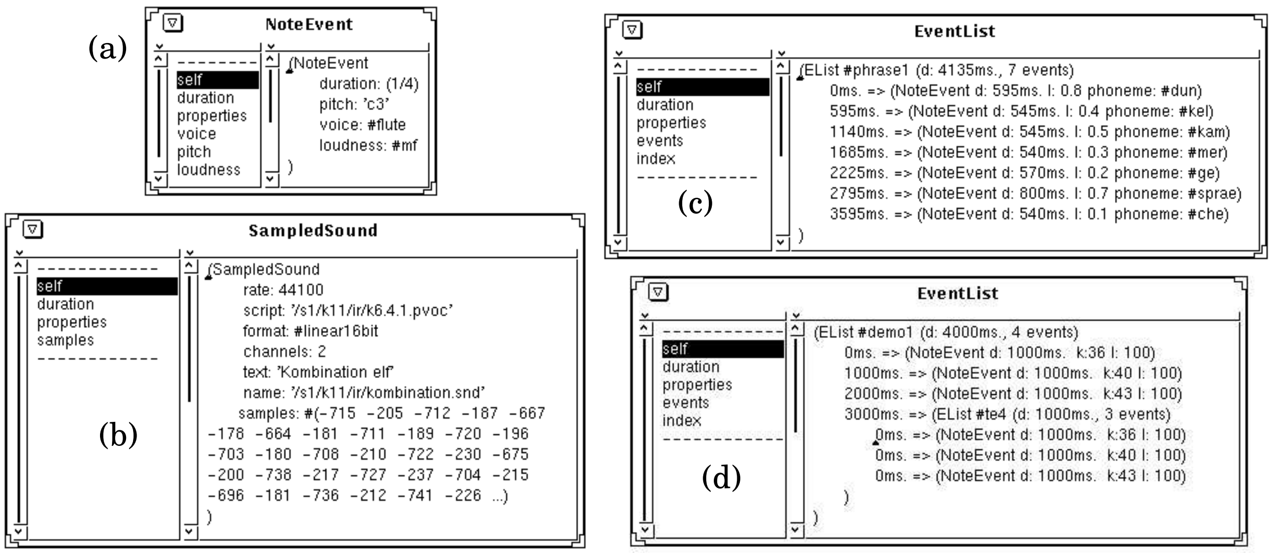
```
“NoteEvent creation examples”
        NoteEvent dur: 1/4 pitch: ‘c3’ ampl: ‘mf’
        (NoteEvent dur: 1/4 pitch: ‘c3’ ampl: ‘mf’) color: #green; accent: #sforzando

        el ← EventList newNamed: #demo1.                                “create a named event list”
        el   add: (NoteEvent duration: 1000  pitch: 36  ampl: 100);      “add an event to it at time 0”
             add: (NoteEvent duration: 1000  pitch: 40  ampl: 100);      “add an event after the first...”
             add: (NoteEvent duration: 1000  pitch: 43  ampl: 100);
             add: (EventList new                          "add a sub-list to it with 3 simultaneous events"
                     add: (NoteEvent  duration: 1000  pitch: 36  ampl: 100)  at: 0;
                     add: (NoteEvent  duration: 1000  pitch: 36  ampl: 100)  at: 0;
                     add: (NoteEvent  duration: 1000  pitch: 36  ampl: 100)  at: 0)
```

Figure 8: Event and EventList usage examples

Event and EventList Inspectors, Editors and Browsers

Figure 9 illustrates the user interface of simple inspectors on MODE events and event lists. The left-hand pane of the inspectors show 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 9(a) is an event inspector showing the internal state of a MIDI-style event (a), a sound object inspector (b), a simple event list in an inspector (c), and a two-level event list where the fourth event is a sub-list (d).



```
(NoteEvent
    duration: (1/4)
    pitch: 'c3'
    voice: #flute
    loudness: #mf
)
```

```
(SampledSound
    rate: 44100
    script: '/s1/k11/ir/k6.4.1.pvoc'
    format: #linear16bit
    channels: 2
    text: 'Kombination elf'
    name: '/s1/k11/ir/kombination.snd'
    samples: #(-715 -205 -712 -187 -667
-178 -664 -181 -711 -189 -720 -196
-703 -180 -708 -210 -722 -230 -675
-200 -738 -217 -727 -237 -704 -215
-696 -181 -736 -212 -741 -226 ...)
)
```

```
(EList #phrase1 (d: 4135ms., 7 events)
    0ms. => (NoteEvent d: 595ms. l: 0.8 phoneme: #dun)
    595ms. => (NoteEvent d: 545ms. l: 0.4 phoneme: #kel)
    1140ms. => (NoteEvent d: 545ms. l: 0.5 phoneme: #kam)
    1685ms. => (NoteEvent d: 540ms. l: 0.3 phoneme: #mer)
    2225ms. => (NoteEvent d: 570ms. l: 0.2 phoneme: #ge)
    2795ms. => (NoteEvent d: 800ms. l: 0.7 phoneme: #sprae)
    3595ms. => (NoteEvent d: 540ms. l: 0.1 phoneme: #che)
)
```

```
(EList #demo1 (d: 4000ms., 4 events)
    0ms. => (NoteEvent d: 1000ms.  k:36 l: 100)
    1000ms. => (NoteEvent d: 1000ms.  k:40 l: 100)
    2000ms. => (NoteEvent d: 1000ms.  k:43 l: 100)
    3000ms. => (EList #te4 (d: 1000ms., 3 events)
        0ms. => (NoteEvent d: 1000ms.  k:36 l: 100)
        0ms. => (NoteEvent d: 1000ms.  k:40 l: 100)
        0ms. => (NoteEvent d: 1000ms.  k:43 l: 100)
    )
)
```

Figure 9: Event, Sound, and EventList inspectors

EventGenerator and EventModifier Usage

The examples in Figure 10 show the usage of the basic event generator and event modifier behavior.

```
“EventGenerator examples: A simple progression—C Major cadence.”
            (EventList newNamed: #progression1)
                add: ((Chord majorTetradOn: ‘c4’ inversion: 0) duration: 1/2);
                add: ((Chord majorTetradOn: ‘f3’ inversion: 2) duration: 1/2);
                add: ((Chord majorTetradOn: ‘g3’ inversion: 1) duration: 1/2);
                add: ((Chord majorTetradOn: ‘c4’ inversion: 0) duration: 1/2)

    “A Trill example—play 2 seconds of 40 msec. long notes on c5 and d5.”
            ((Trill length: 2.0 rhythm: 40 notes: #(48 50))  ampl: 100) play

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

```
“Pentatonic selection that makes a transition from one chord to another.”
    (DynamicSelectionCloud dur: 9
            pitch: #( #(40 43 45) #(53 57 60))       “starting and ending pitch sets”
            ampl: #(80 80 120)                       “static amplitude set”
            voice: #(1 3 5 7)                        “and voice set”
            density: 20)

“Function usage example—make a roll-type eventList and apply a crescendo/decrescendo to it.”
    | temp fcn |
    temp ← EventList newNamed: #test3.               “Create a new named EventList.”
    (0 to: 4000 by: 50) do:                          “Add 20 notes per second for 4 seconds.”
            [ :index |                               “Add the same note”
            temp add: (NoteEvent dur: 100 pitch: 36 ampl: 100) at: index].
    fcn ← LinSeg from: #((0 @ 0) (0.5 @ 1) (1 @ 0)).         “Create a line segment function.”
                                                     “x@y is Smalltalk-80 point creation short-hand”
    temp apply: fcn to: #loudness.                   “Apply the function to the loudness of the EventList.”
```

Figure 10: EventGenerator and EventModifier usage examples

Voices and I/O

The code examples in Figure 11 shows two usages of MIDI output ports and devices for playing a chord, and a formatted cmusic file for output.

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

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

Figure 11: Voice usage examples

The class hierarchies of the MODE’s layout manager and view/editor classes are shown in Figure 12.

```
    LayoutManager (view orientation itemAccessor)—the abstract class
        HierarchyLayoutManager (length xStep yStep treeAccessor)—sends model tree accessing messages
            BinaryTreeLayoutManager—assumes a binary tree
                LeafLinearTreeLayoutManager—lays out leaves in a line
                    TRTreeLayoutManager (sequenceView)—special knowledge of TRTree node types
                TreeLayoutManager—general-purpose tree layout
            IndentedListLayoutManager—lays out hierarchies as outlines
                IndentedTreeLayoutManager (list)—outline-tree layout
        SequenceLayoutManager (timeScale timeOffset)—assumes left-to-right sequential layout in time
            GroupingLayoutManager (levelScale)—shows hierarchical grouping of events
            PitchTimeLayoutManager (pitchScale pitchOffset)—shows pitch as the y-coordinate
                CMNLayoutManager (stepArray staffTop)—approximated common-practise Western


    SequenceViews—display list view where x = time
        PitchTimeViews—SequenceView where y = pitch
            Hauer-Steffens, CMN, other subclasses...
        PhraseViews—spoken utterance editing, derivation of prosidic stress trees
        GroupingViews—display hierarchy of event lists
    SampledSoundEditor
        Mix/Fade/Cross-fade sampled sound files, optionally generate sound file mixing scripts for cmusic
    Location Editors
        PositionTimeViews, Panning EventGenerators and cmusic output
    T-R EventList Editors for Sampled Sounds
        Support for TreeNodes, TRTree creation, derivation and application
```

Figure 12: LayoutManager and View/Editor hierarchies

TR-Tree Editors and Browsers

Two examples of one type of T-R Tree browser view are shown in Figure 13 below (from [Pope 1991c]). 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 <u>sections</u>, <u>phrases</u>, <u>theories</u>, and <u>trees</u>. 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 set of tree types, as shown in the theories list; and there can be many trees within each theory—versions of one 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 home-town 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."), were the "nai" is being exaggerated in length. Editing this tree would change the relative durations of the syllables.

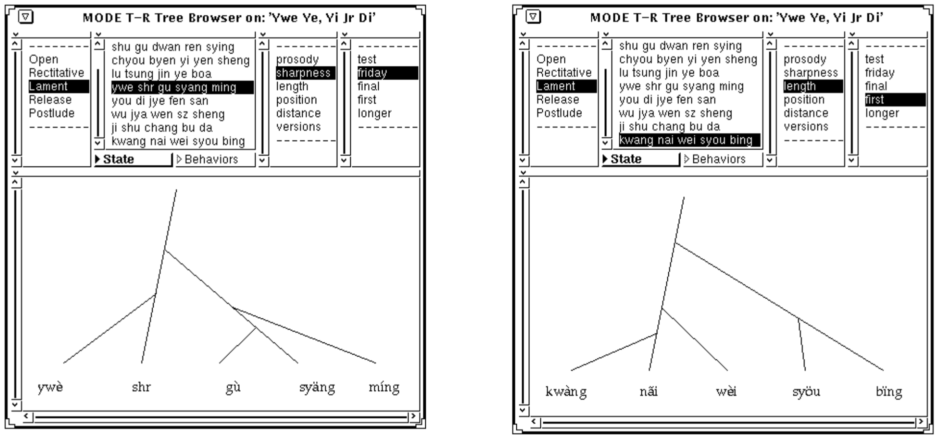
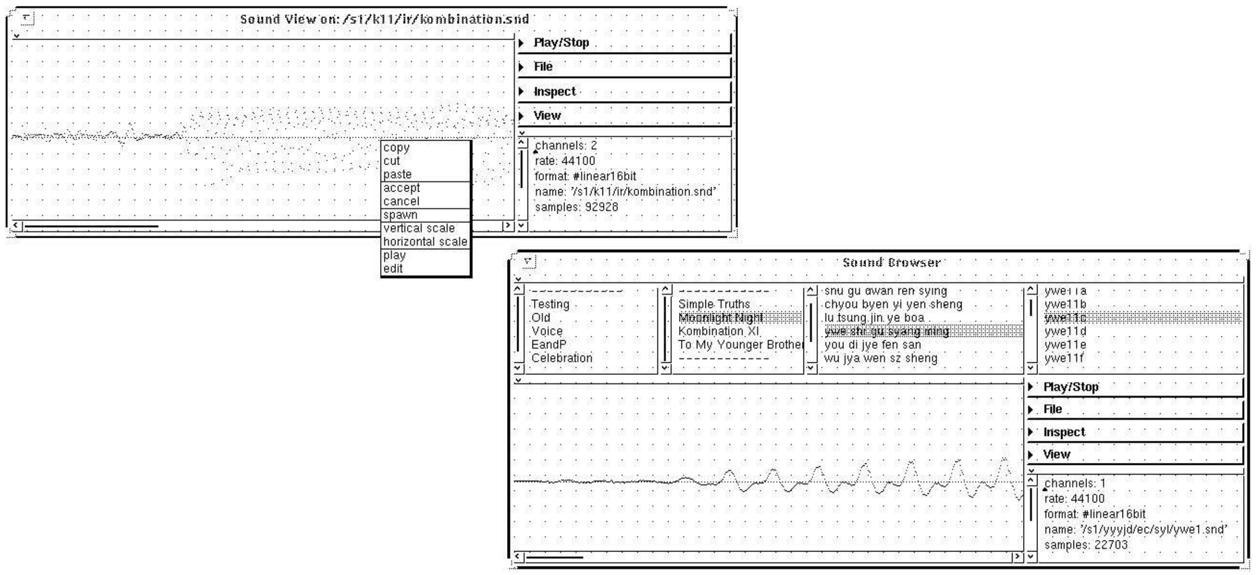


Figure 13: Score Browser/TR-Tree editor examples

Sound Processing, Editing, and Mixing



channels: 2
rate: 44100
format: #linear16bit
name: '/s1/k11/ir/kombination.snd'
samples: 92928

channels: 1
rate: 44100
format: #linear16bit
name: '/s1/yyyjd/ec/syl/ywe1.snd'
samples: 22703

Figure 14: Sound editor and browser

The primary sampled sound interfaces are the sound editor and browser views, shown in Figure 14. The simple sound view supports the basic sampled sound operations shown in the pop-up 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. The lower portion of the browser view is a sound view.

Conclusions and Directions

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 non-commercial distribution via anonymous network *ftp* and can be found in the directory `pub/st80` on the InterNet machine named `CCRMA.Stanford.edu`. There are also PostScript and ASCII text files of several other MODE-related documents there, including an extended version of this paper.

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.

As of the time of this writing (September, 1991), 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). The DoubleTalk re-implementation has yet to be tackled, but should be underway by the time this appears anywhere in print.

The intent of this paper is to present 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 systems to remain stable for a number of years to come.

Acknowledgments

It would be extremely unfair to present the design and/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. I would also like to specifically thank Jan Witt of PCS/Cadmus Computers GmbH, Adele Goldberg and Glenn Krasner of ParcPlace Systems, Inc., and Tony Agnello of Ariel Corp. for their support for (well, toleration of, at least), the on-going development of otherwise quite useless tools for composition, even within the framework of the software development teams of start-up companies.

References

I do not cite the full Smalltalk-80, OOP/music or my own reference lists below; real bibliographies for this project can be found in (Pope 1986) and (Pope 1991b); in order to save space, I cite here only the more hard-to-find references and those not cited in the above collections.

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

Böcker, H-D., A. Mahling, and R. Wehinger. 1990. "Beyond MIDI: Knowledge-based Support for

Computer-aided Composition.” *Proceedings of the 1990 International Computer Music Conference* (ICMC). San Francisco: International Computer Music Association.

Buxton, W. et al.,1978. “An Introduction to the SSSP Digital Synthesizer.” *Computer Music Journal.* 2(4): 28-38. Reprinted in Curtis Roads and John Strawn, eds. 1985. *Foundations of Computer Music.* Cambridge: MIT Press.

Buxton, W. et al.,1979. “The Evolution of the SSSP Score Editing Tools.” *Computer Music Journal.* 3(4): 14-25. Reprinted in Curtis Roads and John Strawn, eds. 1985. *Foundations of Computer Music.* Cambridge: MIT Press.

Cadmus GmbH. 1985. “Cadmus Computer Music System Product Announcement.” *Computer Music Journal.* 9(1): 76-77.

Computer Audio Research Laboratory. 1983. *CARL Software Startup Kit*. San Diego: CARL, Center for Music Experiment, University of California at San Diego, December, 1983.

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*. Report CSL-80-10. Palo Alto: Xerox PARC.

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.* Report SSL-76-1. Palo Alto: Xerox PARC

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

McCall, K. 1980 *The Smalltalk-76 Music Editor*. Xerox PARC internal document.

Pope, S. T. 1982. “An Introduction to *msh*: The Music Shell.” *Proceedings of the 1982 International Computer Music Conference* (ICMC). San Francisco: International Computer Music Association.

Pope, S. T. 1986. “The Development of an Intelligent Composer’s Assistant: Interactive Graphics Tools and Knowledge Representation for Composers.” *Proceedings of the 1986 International Computer Music Conference* (ICMC). San Francisco: International Computer Music Association.

Pope, S. T. 1991a. “Introduction to the MODE.” in (Pope 1991b)

Pope, S. T., ed. 1991b. *The Well-Tempered Object: Musical Applications of Object-Oriented Software Technology.* Cambridge: MIT Press.

Pope, S. T. 1991c. “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’).” *Proceedings of the 1991 International Computer Music Conference* (ICMC). 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.

Toenne, A. 1988. *TRAX Software Documentation*. Dortmund, Germany: Georg Heeg Smalltalk Systeme GmbH.