HMSL (Hierarchical Music Specification Language): A Theoretical Overview

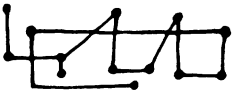
Larry Polansky; Phil Burk; David Rosenboom

Perspectives of New Music, Vol. 28, No. 2 (Summer, 1990), 136-178.

Stable URL:
http://links.jstor.org/sici?sici=0031-6016%28199022%2928%3A2%3C136%3AH%28MSLA%3E2.0.CO%3B2-7

Perspectives of New Music is currently published by Perspectives of New Music.

HMSL (Hierarchical Music Specification Language): A Theoretical Overview



Larry Polansky
and
Phil Burk
with
David Rosenboom

Preface

This article provides an overview of the computer music language HMSL, and some brief examples of its use in the compositions and experiments of Phil Burk and David Rosenboom. HMSL itself is written by Burk, Polansky, and Rosenboom.

HMSL Overview

Brief description, history, and technological considerations

HMSL is an object-oriented computer-music programming language written at the Center for Contemporary Music, Mills College, Oakland. HMSL was begun in 1980, and is still being developed. The language is completely extensible, and is distributed in fully documented source code so that users may alter and extend it to suit their own needs. Briefly, the HMSL environment consists of:

1. a general-purpose object-oriented programming language (ODE) in which HMSL itself is written, and which is fully accessible to the user;
2. a completely general hierarchical scheduler (PE), capable of scheduling processes, events, and data of all sorts;
3. a large set of flexible data structures, specifically designed for musical artificial intelligence and real-time musical performance, which can be extended and redesigned by the user;
4. a flexible and customizable stimulus-response environment that allows the user to design complex and intelligent real-time interactive performance situations;
5. a *Virtual Device Interface*, which is a set of utilities for user specification of output data, allowing the scheduler, stimulus-response environment and other parts of the system to be completely independent of eventual sonic (or other forms of) realization;
6. several graphic interfaces for creating, editing, and interacting with data and processes, and a graphics language for creating one's own graphic interfaces;
7. low-level device drivers for MIDI, sound-synthesis, and graphics applications;
8. a *Score Entry System* for entering data in more conventional musical ways;
9. a large set of miscellaneous programming utilities and libraries designed for musical applications.

HMSL's design intent was to provide a powerful and flexible real-time programming environment for music. Its authors were particularly interested in a user-defined stimulus-response environment for performance, in the specification of large-scale hierarchies, morphological transformations (hierarchically), and the ability to interface the system to any external hardware, including but not limited to sound-producing devices.[1] HMSL was designed to be highly portable, to run on the new series of 16-bit

microprocessors that had just been introduced (around 1980). The Motorola 68000 series was chosen as the host processor, and Forth was chosen as both the operating system and the programming language in which to write HMSL.[2]

In the interests of portability and availability, HMSL was targeted for small computer-based systems. The Mills College Center for Contemporary Music has traditionally been an important focus of research and development in live electronic music and, consequently, in affordable technologies.[3] HMSL was intended as a general-purpose and highly flexible real-time music language for performance that would be a "next step" in the evolution of live electronic music.

FORTH AND ODE

Forth has been for some years the language of choice for many composers working in microcomputer music.[4] It is particularly useful and applicable in real-time computer-music applications because of its flexibility in hardware control, conservative memory requirements, and extensibility. Forth code is highly portable, and one of our interests was to have in HMSL a kind of "clearing house" of musical software ideas that could be shared, added to, and learned from by a wide community of users.

HMSL has a "three level" environment for user programming: Forth, ODE, and HMSL. Phil Burk wrote ODE (Object Development Environment) in 1986 to further develop HMSL's data structures. HMSL's data structures are now completely written in ODE, and users have full access to that environment as well as Forth for their own code. All three "levels" are of course concomitant, and a typical HMSL program uses all of them most of the time.[5]

ODE

Burk's ODE is robust and full-featured, and was designed for real-time programming. Those familiar with object-oriented programming languages will find ODE to be reasonably standard. ODE is loosely based on the Smalltalk-80 model of object-oriented programming, and includes the concepts of classes, objects, inheritance, message passing, methods, and instance variables. Much of the syntax for Burk's ODE is based upon the NEON language for the Macintosh, an earlier object-oriented Forth system (from Kriya Systems).

Object-oriented programming environments are attractive to musicians because they allow composers to describe musical events in very broad

terms, and attribute to them certain characteristics and behaviors in much the same way that composers treat conventional musical phenomena. For example, the "row" in serial music, the "pc set" in atonal music, or the "subject" in a Baroque fugue are often considered independent entities, with certain "self-transformative" properties (such as the tonal answer for the fugue, or the canonical transformations for a row), measures (e.g. Forte's "interval vector" for a pc set), and attributes (e.g. length, key, et cetera). These measures, attributes, and transformations are universal over the set of musical objects of a given category, but may have different local values for a specific instance. This way of describing musical events is well suited to a programming style that depends on the definition of self-contained entities with specific attributes and the ability to communicate with each other. Object-oriented musical programming, it should be noted, offers the composer far more than a means of imitating preexistent musical formulations. It is a powerful language for describing new and experimental musical entities.

There is extensive literature on object-oriented programming and, particularly, object-oriented music environments (Cox 1986; Goldberg and Robson 1983; Pope 1986, 1987, 1989; Flurry 1988; Polansky, Burk, and Rosenboom 1987; Burk 1987; Scaletti 1987, 1989, 1989a). Burk's description of and tutorial for ODE in the *JForth Professional* version 2.0 manual (Burk et al. 1989) is a comprehensive description of this environment.

ODE FEATURES

Although conceptually similar to Smalltalk, many of ODE's features look different because it is Forth-based. Unlike Smalltalk, low-level data structures (like constants, variables, numeric literals, simple functions, and so forth) are not objects, they are simply Forth words. In general, ODE is used only to implement higher-level data structures.

ODE is optimized for real-time music performance. Unlike Smalltalk, there is no "garbage collection" scheme (for a description of this in Smalltalk, see Goldberg and Robson 1983, 674–85). Message passing in ODE is compiled directly to machine language (except in the case of *late-binding*). Two examples below will illustrate some of ODE's particular syntax, as well as some interesting musical features.

Late-binding. An important feature of ODE is *late-binding*, the ability for a method to pass a message to an "unspecified" or variable object.

For example, consider a class called `SCALE` with specific instances called `MY-SCALE` and `YOUR-SCALE`. The `SCALE` class has methods defined for using

various tunings. The following would be the usual way of telling an instance of the SCALE class which tuning to use ("early-binding"):

```
USE.SLENDRO: MY-SCALE
```

Late-binding allows the programmer to leave unspecified the actual instantiation used, by passing the address of that instance to the method via the Forth stack, as in the following:

```
VARIABLE CURRENT-SCALE ( declare ''holder'' variable )
```

and then somewhere later in the code:

```
YOUR-SCALE CURRENT-SCALE ! ( make YOUR-SCALE current by )
                           ( storing it in the variable )
```

and then, somewhere else again in the code:

```
CURRENT-SCALE @ USE.SLENDRO: []
```

This last line is read as "Current scale fetch, use-slendro, late-bind," where CURRENT-SCALE is a variable containing the address of some object. "@"is the Forth word ("fetch") for retrieving the contents of that variable. "!" is the Forth word ("store") for storing to a variable. The point of the above procedure is that we know we want to use slendro, but we don't know which scale ("yours" or "mine") we will be using at that point in the program.

Dynamic Instantiation. In ODE, objects are usually instantiated at compile time, and given a name. For example:

```
OB.HARD.LICK FUNKY-LICK
```

creates an instance of the class OB.HARD.LICK called FUNKY-LICK. In ODE this is called *compile-time instantiation*. *Dynamic instantiation*, however, allows the composer to create objects "on-the-fly" in response to stimuli (such as MIDI or analog input), or algorithmically. In many cases, particularly musical ones, the programmer may be interested in the *N*th object in a list of many, more or less identical objects, rather than a single named object. Dynamically instantiated objects are referenced by their address, not their name. Their addresses can be maintained in a list of many objects of the same class, and referenced by indexing into that list.

In the following example, an object of a class is instantiated, and its

address, left on the stack by the word INSTANTIATE, is placed in a variable for later use:

```
VARIABLE CURRENT-REAL-COOL-LICK
INSTANTIATE OB.HARD.LICK
  ( -- address-of-object-instantiated, from the )
  ( predefined class called OB.HARD.LICK )
CURRENT-REAL-COOL-LICK !
  ( store object's address in variable )
```

and later, elsewhere in the code:

```
COOL-LICK-NEEDED?
IF
  ( check to see if cool-lick-need is true? )
CURRENT-REAL-COOL-LICK @ PLAY: []
  ( play the current one )
THEN
```

This technique makes the creation of large lists of musical data structures quite easy. In live performance, it allows the computer to create, transform, and keep track of complex sets of musical entities without the composer/performer specifying them one by one.

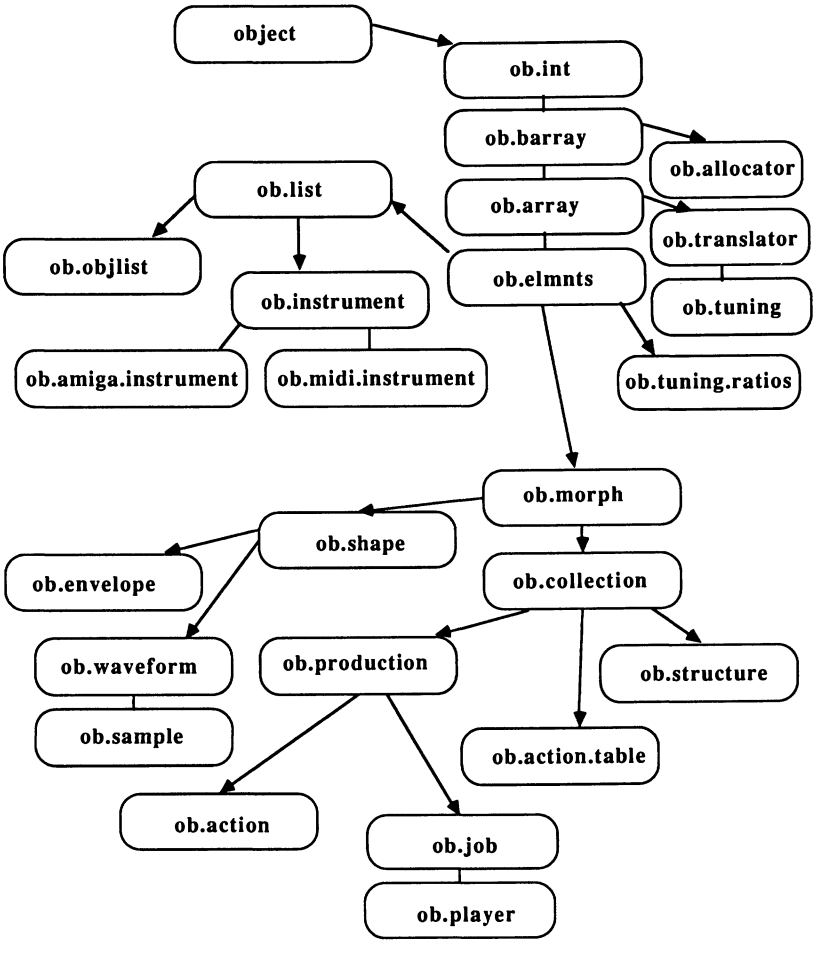
HMSL Data Structures

This section briefly describes some of the basic musical data structures in HMSL. For a fuller description of the methods, uses, and intelligences of each data type, the reader is referred to the appropriate section of the HMSL manual.

MORPHS

The fundamental object in HMSL for the creation and execution of musical heterarchies[6] is the *morph*, short for *morphology*. Morphs are holders of ordered data which can communicate with each other and with the HMSL scheduler. Subclasses further down the inheritance tree from morphs have more specific musical intelligences (see Example 1).

Since one of the music-theoretical and compositional design motivations for HMSL was to develop an environment for the manipulation of generalized morphological constructs, this rather general notion of an "ordered



EXAMPLE 1: HMSL CLASS INHERITANCE DIAGRAM

data" holder became the kind of "ur" class. It is not very different from an *array*, and is actually a subclass of *arrays*. What distinguishes morphs, and what makes them musically useful in HMSL, is that they have some concept of "execution"—they can be "played" or "scheduled," and are ordered lists of data which "do something." Integral to this notion of "execution" are methods for communicating with "parent" and "child" morphs, such as methods for "starting a child," or for "telling a parent" morph that the "child" morph is "done" executing.

There are various methods for adding, deleting, and editing data for all morphs. Most of the HMSL musical data types (shapes, collections, structures, productions, jobs, et cetera) are subclasses of the morph class.

SHAPES

Shapes, a subclass of morphs, are multidimensional arrays with methods for manipulating data in musically useful ways. In HMSL shapes can contain "raw" musical data of any type: a parameterized list of musical events (e.g. notes, with the dimensions of duration, loudness, and pitch, and so forth), complex arguments to some function (like filter coefficients to an outboard device, perhaps with associated durations), or a list of data specifying statistics of musical form at any level. Although shapes may contain pointers to other morphs, they can not heterarchically contain other morphs. Shapes are generally the lowest node in an HMSL heterarchy. The concept of order is implicit in the definition of this class, and the *i*th "multidimensional" point in a shape is referred to as the *i*th *element*.

For example, the values in a three-dimensional shape might specify the mean, range, and standard deviation of pitch in some temporal slice of a piece. This shape would then be used by another HMSL object to generate data. One can also use shapes to store pointers to larger musical data types, such as other shapes, and then apply various transformative procedures to the "holder" shape to reorder its component shapes.

The shape class contains many methods for manipulating, editing, storing and retrieving values. Users often write simple extensions to this class to implement a new method. The shape class, while musically powerful, is intentionally a bit austere in the type of morphological manipulations provided. These are more or less limited to editing operations, transposition, inversion, reversal, deletion, addition, replacement, scramble, randomization, and a few others. All of these operations have several associated parameters, such as the range of values in the shape itself, which dimension to affect, axis of inversion, range of randomization, and so on.

Users can easily write new methods using the tools provided by HMSL in the preexisting methods for the class. To define a new class, a programmer can, for example, simply inherit everything from the shape class, and write one or more new methods. HMSL is not intended as a library of musical utilities, but rather as an environment for creating one's own libraries. Users write new subclasses of the shape class to include their favorite algorithmic procedures.[7]

In HMSL, shapes serve as a fundamental representational construct for raw data that may include alternative representations of melodic data as well as high-level formal descriptions of musical processes (Xenakis's UPIC system is an interesting example of the way that simple descriptors can generate an extraordinarily wide variety of meanings [Lohner 1986]). Future versions of HMSL may incorporate other powerful descriptors of morphology as standard features, such as the description of morphologies as pure "contours," algorithmically in terms of distribution functions, or

by some standard set "descriptor" (such as Forte's interval vector). These are all cases where one signifier, for example a given contour, or a given interval vector, could have many realizations, and it is interesting to consider how the HMSL environment might realize the specifics of these modes of morphological specification.

COLLECTIONS

If `shapes` are described as holders of "raw data," *collections* may be described as holders and executors of "raw" `morphs`. Collections, the primary heterarchical units of HMSL, are morphs that can contain any other morph in any "arrangement." Collections can be treated (i.e. executed, put in other collections, transformed) as one entity, so that large scale formal and scheduling processes are facilitated. Since collections can include other collections (or structures, players, jobs, and so on) HMSL is capable of any level of heterarchical complexity.[8] These complex heterarchies might be thought of as tree structures in the conventional sense, with any node on the tree being another artbitrarily nested tree.

Collections have execution "intelligence" called *behaviors*. Behaviors are written by the user, although several defaults are supplied in HMSL. The system-supplied *parallel* and *sequential* behaviors execute the component morphs of a collection either sequentially or in parallel. HMSL supplies other behaviors as well, such as a *random sequential behavior*.

User-written behaviors may select any combination of a collection's children for execution. These behaviors are quite powerful in the construction of complex large-scale musical forms, such as those in which the actual form of a work is dynamically changed by real-time input. (This idea is used extensively in Polansky's composition *Cocks Crow, Dogs Bark*... [Polansky 1988]). Other behaviors might check the status of stimuli, alter the configurations of heterarchies within the system, change variables, generate morphological data, instantiate objects, or load new functions or pointers in real time.

Additional features in collections include: *repeat-count* (how many times to execute their components in a given execution of the collection itself), *weight* (an instance variable which is often used by the system to compute the probability of executing a given collection), and various methods for editing and maintaining component morphs. Collections may contain user-written functions (in the form of Forth/ODE/HMSL routines) that are executed when a collection is begun (*start-function*), repeated (*repeat-function*) or terminated (*stop-function*). Other methods allow for customized scheduling within HMSL.

Since collections are a subclass of morphs, they inherit all the basic

features of the basic morphological paradigm of HMSL. Most other data types are a subclass of collections (like *productions*, *jobs*, *structures*, and *players*). Attributes like repeat-count and weight are inherited more or less universally throughout the system.

STRUCTURES

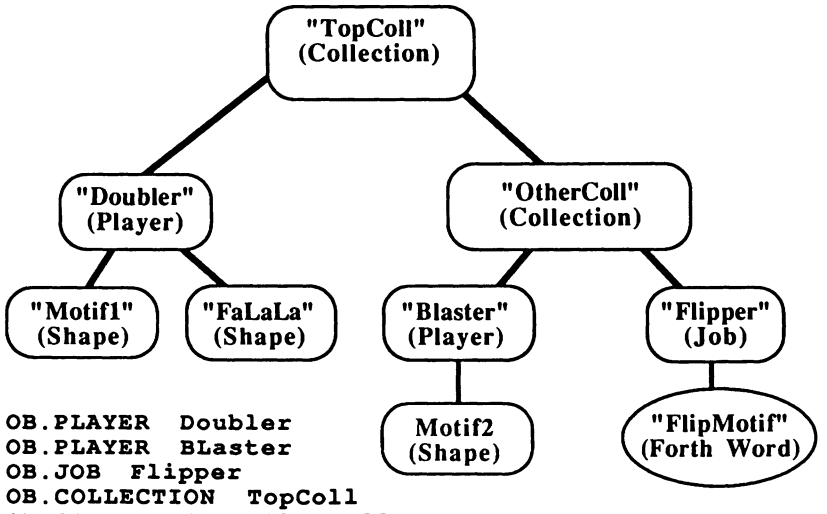
Structures are a subclass of collections, with one added feature, a *transition matrix*. For a structure with N components, this matrix is $N \times N$ large, with several methods for maintenance and editing. One of the simplest behaviors for a collection is to use the weights of component morphs to decide the likelihood of execution—structures allow the user to combine this with a probability that one component will be succeeded by another (in HMSL this is called a *tendency*). These two values are often used together in the determination of execution order in a structure. Structures may be thought of as simple first-order Markov chains, but the associated matrix could have many uses (as has been demonstrated by graduate students at the Center for Contemporary Music[9]). An additional HMSL class, provided on disk as an optional library, is the *Markov* class, which provides useful methods, such as orthogonalization and selection of highest probability, for pieces that utilize Markov processes.

PRODUCTIONS

Productions are the simplest data structure in HMSL: morphs that contain a list of user-written routines. They provide a simple way of heterarchically scheduling and executing algorithms, as opposed to only data (as in collections and structures). Productions contain only a list of routines, and can contain no other morphs. They are a subclass of collections, and inherit all the parent class's methods and features (repeatability, weight, editing, and so forth).

A production consists of a list of pointers to executable addresses of Forth words, which may be written in some combination of HMSL, ODE, Forth, assembler, and so on. The code inside a production may reference and alter any part of the system (as in behaviors), and can even insert new routines into the list of the production being executed. For example, this might occur in response to a given key being pressed on a keyboard, or to some condition set by software. This is a feature which can be quite powerful in musical performance situations. Productions can also be used to change the weights of other morphs, execute other morphs, transform a shape, alter variables, and so on.

Productions are useful and powerful in HMSL, as a kind of all-purpose utility for scheduling software events. The choice of the term "production" was originally derived from formal language theory, where it refers to a set of rules that specify how morphologies (in HMSL, most often shapes) might be created and transformed.[10] Example 2 illustrates the way many of HMSL's data structures can be combined into a heterarchy.




```
OB.PLAYER  Doubler
OB.PLAYER  BLaster
OB.JOB  Flipper
OB.COLLECTION  TopColl
OB.COLLECTION  OtherColl
STUFF{  Doubler  OtherColl  }STUFF:  TopColl
STUFF{  Motif1   FaLaLa     }STUFF:  Doubler
STUFF{  Blaster  Flipper    }STUFF:  OtherColl
STUFF{  Motif1  }STUFF:  Blaster
STUFF{  'C  FlipMotif  }STUFF:  Flipper
```

EXAMPLE 2: AN EXAMPLE HETERARCHY

"TIMED MORPHS:" JOBS AND PLAYERS

An important aspect of collections and productions is the way they deal with time, which is not at all. Once one of these morphs is executed, things happen "as fast as possible," for these morphs have no associated notion of duration, except for their start and repeat delays. Time in HMSL is left unspecified until a very low level of data specification is reached. Shapes, which cannot be executed by themselves (they must be used inside another morph), have no dimension fixed as time, so that the dimensions of a shape might be reinterpretable at various times and by various parts of the system,

in radically different ways. A common HMSL experiment is to use a small number of shapes, or just one, as the source of all morphological "data"—reinterpreting that data in a variety of unusual ways.

There are two special subclasses of collections associated with actual time values: *jobs* and *players*. Both of these morphs are said to be "tasked" by the system, because once they are executed, the scheduler continually checks to see if they need to do something (that is, if a certain duration has elapsed). This is in contrast to collections, which, once executed, may be "forgotten about" by the scheduler until they signal that they are "done."

Jobs. Jobs are the simpler of the two morphs. They are similar to productions in that they consist primarily of a list of Forth routines, but they also have an associated duration, which specifies how often their list of functions will be executed once the job is "made active" (executed). This duration can be changed in real-time by the job itself, or by any other part of the system; it need not be static.

A job must "stop itself"—once it is executed, it will keep on going until it decides (by a simple protocol) that it is finished. It is also possible for other morphs and software processes to cause a job to end. Jobs are excellent for use as background, scheduled tasks. Even though jobs are among the simplest morphs, they are by far the most used, and are, fact, similar to what programmers normally think of as a "process," or "background task." They also have a few other methods associated with them which allows for more sophisticated interaction with the *Virtual Device Interface* (see below), customized scheduling, repeats, and so on.

Players. Players are a more complex, scheduled morph. They are the most usual means by which shape data is interpreted in time for sonic or other output. Players consider one dimension of their component shapes as duration. Once executed, players send, at the appropriate times, the current element number of a component shape to an associated *instrument.* The instrument may interpret the multidimensional data of this element in any way. A kind of three-level process typically occurs in HMSL:

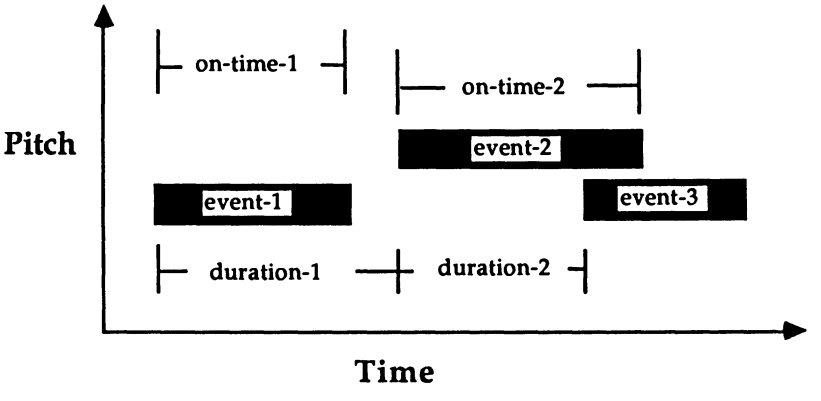
```
RAW DATA → TIMED RAW DATA → INTERPETED TIMED DATA

(SHAPES)  (PLAYERS)          (INSTRUMENTS)
```

Players are a sort of "middle management" for general data, allowing that data to be "patched" to specific interpretations in time. This three-level approach is highly flexible since the patch itself, and many of its details (e.g. which dimension to use for duration, offsets, the order of shapes in the player's list, the instrument in use, and so on) can be changed rapidly in

real time from any other part of the system (by jobs, stimulus-response, productions, and so on). Players may also bypass the "duration dimension," and use a specified duration function (user-written) for the scheduling of their data.

Duration is usually defined very specifically in HMSL as the elapsed time between the *beginnings of events*. This is different from what might be thought of as *on-time*, that is, the elapsed time of an *actual event*. This concept of on-time can be useful, for example, in the specification of MIDI note events, which have a specified elapsed time between "note-on" and "note-off" (see Example 3).[11]



EXAMPLE 3: "DURATION" VS. "ON-TIME"

HMSL also includes the concept of *duty cycle*: the ratio of on-time to duration of an event. In players, there are various ways for the user to specify and manipulate the duty cycle. One is to specify a ratio of the entire player, such as 4/5, which tells the player to "turn-off" the event (that is, send an "off" message to its instrument) after eighty percent of the current duration has elapsed. Nothing will happen in the following twenty percent of the specified time interval, and then an *on message* will be sent to the instrument with the corresponding values for the next element of the player's shape. To achieve polyphony, the user may specify a ratio of greater than 1, which would mean that an event would be turned off after the "next" one was turned on. The user may also specify a given dimension of the player's shape to use as the actual on-times for each corresponding element of the shape. In this way, each element of the shape can have a different duty cycle. This may be edited graphically in the shape editor, or subjected to the usual software manipulations and controls.

There are several more sophisticated ways to specify the timing of events in players, particularly with regards to polyphonic events. This includes the specification of *absolute start times* for the occurrence of events, rather than the conventional specification of *relative start time* as described above. (Absolute start times are cumulative times from the beginning of execution of the player.)

Execute, Create, and Perform

Conceptually, the functions of HMSL are divided into three main modes: *Execute*, *Create* and *Perform*. These are not precisely defined terms, but ways of describing various features of the system. *Execute* describes how the data structures are scheduled, sent to hardware, and interact with each other and the user in real time. *Create* describes methods of input and editing. *Perform* describes how stimulus-response mechanisms can be added to the system, to coexist and interact with the other two modes.

EXECUTE

The main mechanism of Execute, and the central scheduling intelligence of HMSL, is the *Polymorphous Executive* (PE).[12] The PE is a sophisticated "round-robin" scheduler, which keeps track of all *active objects* in the system in their heterarchical complexity, and tries to accurately schedule and execute all timed events. It has no theoretical limitations on its complexity, but since it is essentially a multitasker running on a single processor, there are practical limitations on how much the system can handle.

The PE works in a straightforward, message-passing manner. Morphs execute their children by sending an EXECUTE: message to them. When those children are finished they send DONE: messages back to their parents, all the way up the tree. Posting, executing, and management of the active object list is generally done by the morphs themselves. The methods for doing this are relatively simple, and the tools are provided for the composer to invent unusual scheduling and execution interactions between morphs (this has been exploited by advanced users of the system, often with unexpected results).

Software Scheduling. With the exception of MIDI and Amiga local sound output, which use an interrupt-based event-buffering scheme, HMSL uses *nonpreemptive scheduling*. Partly motivated by the complex data structures of the system, it also provides tools for the user to actually alter the fundamentals of HMSL's scheduler, and to make it a flexible and compositionally

rich aspect of the environment. Since the scheduler is itself written as part of ODE, the user can easily manipulate the very mechanics of HMSL's real-time intelligence. For example, the user might restructure the *active object list* itself in response to stimulus, or manipulate system time, and so on—following the philosophy that HMSL should be as open to user innovation as possible.

Event Buffering and Time Deformations in HMSL. In scheduling complex musical events, time deformations can often be caused by an event performing time-consumptive calculations. For this reason, all MIDI and Amiga local sound ouput in HMSL is *event-buffered* and interrupt-driven in a manner similar to that of the language Formula (Anderson and Kuivila 1986, 1988, 1989). This provides for highly accurate timing in which time deformations are seldom an issue. The HMSL scheduler runs "ahead" of the real time clock, allowing complex events to be precalculated before their output is needed. The resulting low-level output events are stamped with the *virtual time* of their desired time of occurrence, and stored in the *event buffer* until then. By manipulating virtual time (through routines supplied by HMSL) the user can perform complex scheduling of low-level events. Other forms of data may also make use of event buffering, but at present only MIDI and Amiga local sound data is event buffered by the system.

HMSL has other ways of dealing with time deformations of non-MIDI events, allowing for highly customized response by players and jobs to these deformations. Various scheduling algorithms are available to each individual player or job that deal with time deformations in different ways (referred to as *epochal* and *durational scheduling*; see the HMSL Manual for a complete description).[13]

In HMSL, either a hardware or software clock may be used. Simple routines allow the user to query, reset, and set the resolution of the hardware clock at any time, allowing full access to the internals of the entire system's scheduling mechanisms. Use of the software clock requires users to increment the clock themselves. This allows for easy synchronization to other systems, or the performance of complex global algorithmic operations on heterarchical scheduling. HMSL also has facilities for synchronization with an external MIDI clock or MIDI time code.

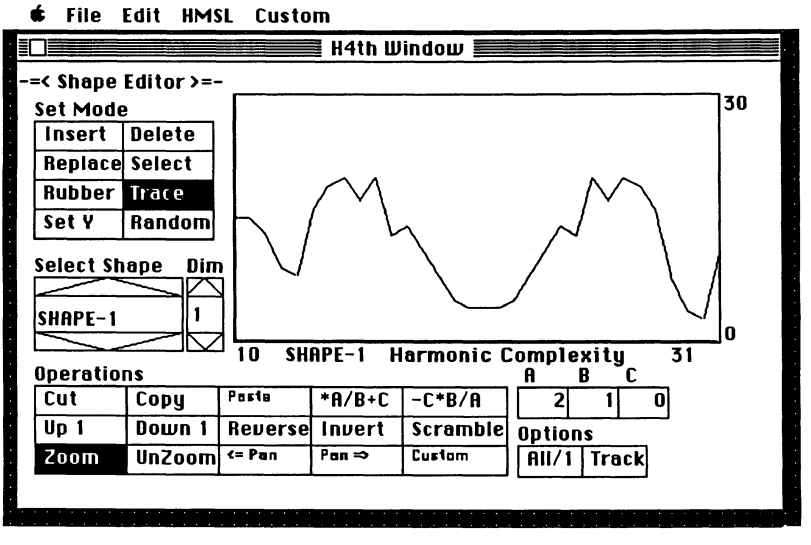
CREATE

HMSL is a "WYSWYH" ("what you see is what you hear") environment, and implements graphic user interfaces whenever possible. There are several screens that allow the user real-time graphic editing and control of data and functions. All of the HMSL graphics are based on an object-oriented graphics library written inside of HMSL.

The user has full access to the graphics library, both at the lowest (draw a line, text, mouse tracking, and so on) and higher levels (various graphic objects and screens) and may also create her own custom graphics. The parent class of graphic objects is called *controls*. Subclasses include *menu grids*, *check grids*, and *radio grids*, and classes for mouse-controlled numerical values (such as *faders* and *counters*). The user may design control grids and customize HMSL's interaction. Controls may be placed in *screens* (called *Custom screens*), and the user may create these as well.

Three predefined graphics screens are supplied by the system: the *Shape-Editor* screen, the *Action* screen, and the *Sequencer* screen. These, along with user-created *Custom* screens, are called from an HMSL menu.

The *Shape-Editor* screen allows the user to input and edit shapes with a variety of functions, implementing most of the shape class methods. Users can cut, paste, copy, replace, scroll, zoom, transpose, reverse, invert, scale, draw, randomize, and so on. The user's own custom transformations may be added to this screen as well. All editing of shapes is in real time, so that if the shape is being played by the PE, changes will be heard as they are made. (Example 4 shows the editing of dimension 1, which contains information regarding "harmonic complexity" of a shape called SHAPE-1.[14] The user has selected a specific range of values in SHAPE-1 to edit).



EXAMPLE 4: SHAPE EDITOR SCREEN

The *Perform* screen is described below in the section on the *Perform* environment. All HMSL screens (the *Shape-Editor*, *Sequencer*, *Action*, and *Custom screens*) are accessible to the programmer; by using the graphics

library, the user can make overlays, additions, or alterations of the screens provided. (The Amiga version of HMSL also includes tools for integrating video and animation routines.)

The *Sequencer* screen resembles a typical MIDI sequencer, but all of its functions may be customized within HMSL. Typically, it can be used for 16-track recording into HMSL shapes, but MIDI input may be redirected and redefined in any way in real time, and of course, values recorded into shapes via this screen may be used to control any aspect of the system.

PERFORM

Perform is the primary mechanism for *stimulus-response* interaction in HMSL. The *Perform* environment consists mainly of two data structures: actions and the *Perform* screen.

Actions. Actions[15] are a subclass of productions, and are the basic unit in custom stimulus-response events. An action consists of a *stimulus* and a *response*, user-defined Forth routines that must obey a simple protocol: the stimulus must leave something on the stack, and the response must take something from the stack. Usually, the stimulus leaves a flag for the response to use in conditionally deciding whether to execute.

The following is an example of a simple action definition, with line numbers inserted at the beginning for purposes of explanation below.

```
1) OB.ACTION MY-ACTION
2) : MY-ACTION.STIMULUS
     ( -- flag, true if C, false if other note )
3)   MIDI.LAST.KEY? ( -- key-value )
     12 MOD ( -- key-value-mod-12 )
     0= ( -- true|false )
4) ;
5) : MY-ACTION.RESPONSE  ( flag --, execute if true )
6)             IF
7)             TIME@ 0 EXECUTE: MY-COLLECTION
     ( at this point,one could also use a simpler method, )
     ( START: MY-COLLECTION )
8)         THEN
9) ;
10)     'C MY-ACTION.STIMULUS PUT.STIMULUS: MY-ACTION
11)     'C MY-ACTION.RESPONSE PUT.RESPONSE: MY-ACTION
12)    MY-ACTION PUT.ACTION: ACTION-TABLE
```

Comments on the above code:

(1) Instantiate an action called MY-ACTION.

(2) Start (with a ":") the definition of a Forth word called MY-ACTION.STIMULUS, which will be the stimulus "field" of MY-ACTION.

(3) Define the word MY-ACTION.STIMULUS. MIDI.LAST.KEY? is another Forth word which we will assume to have been written already, which returns the value of the last key struck on a MIDI keyboard and then resets it to avoid multiple triggerings (see the section on the MIDI Library below). The modulo 12 of that key value is taken, so that if it is a MIDI "C" or octave of C (12, 24, 48, 60, 72), the Forth word " = " will return a "true" value. If the MIDI key is any other value, a false will be left on the stack.

(4) End the definition of the Forth word (";").

(5) Begin the definition of the Forth word MY-ACTION.RESPONSE, which will be used in the response "field" of MY-ACTION.

(6) Test the flag on the stack at entry. If true, execute the code between the IF and the THEN.

(7) Execute the job MY-COLLECTION, using the current time as the starting time (TIME@ returns the current system time), and the value "0" as an *invoker*, meaning that no other morph invokes MY-COLLECTION—it is the highest level in its "tree." Note that this method EXECUTE:, is used internally as well by the PE for the heterarchical execution of morphs, and usually the parent morph is passed as the invoker to the child morph. This is a good example of one simple way the user can actually manipulate some of HMSL's internals. For example, one could extend this line of code slightly by writing:

```
MIDI.KEY? 12 / 60 * TIME@ + 0 EXECUTE: MY-COLLECTION
```

which would take the MIDI key number, divide by 12, and use that value (the "octave number") as a "delay value" in seconds (multiplying by 60, which is the number of ticks per second) for the execution of the collection after the key has been hit. In other words, the lower the key, the shorter the delay before the collection is executed. Alternatively, for the same effect, the user could also simply pass this value to the START-DELAY of MY-COLLECTION.

(8) THEN ends the IF . . . THEN construct. This might seem unusual to, say, "C" programmers. Forth is RPN, or reverse Polish notation—which means, in this case, that the code executed in the "true" case comes before the THEN.

(9) End the definition of the Forth word `MY-ACTION.RESPONSE`.

(10) Use the method defined for the action class, `PUT.STIMULUS:`, to place the Forth word `MY-ACTION.STIMULUS` in the action called `MY-ACTION`. `'C` is a Forth word which returns the executable address of the following word.

(11) As in (10), but for the response word.

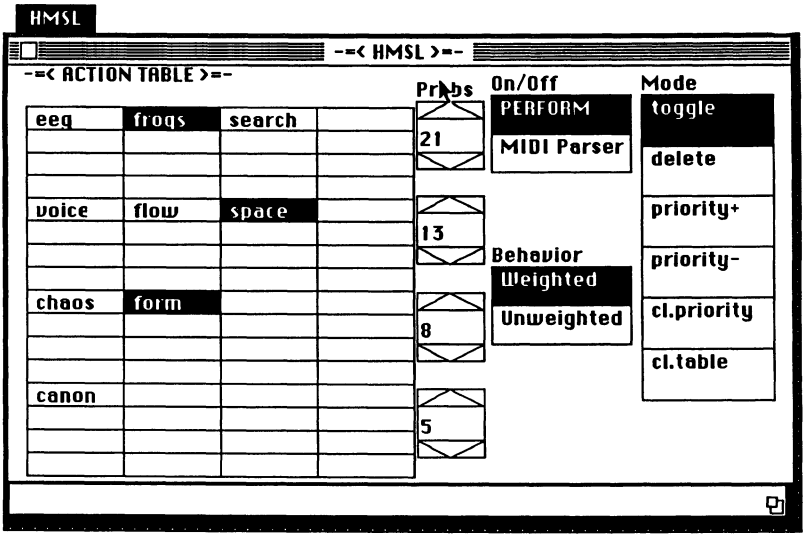
(12) Place the action `MY-ACTION` in the *action-table* (see below), so that it may be accessed on the *Perform* screen in real-time.

This example shows how actions are generally created. The stimulus and response can be any executable routines, can be quite complex, and can reference any other part of HMSL, including the current action and all other actions. In this case, the stimulus comes from someone hitting a keyboard, but "virtual stimuli" such as software-generated events are equally possible.

Actions have many other utilities for experimentation in stimulus-response interaction, including local and global counters, arguments for stimulus and response, fields for initialization and termination functions (Forth words that will execute when action is turned "on" or "off" in software or by the user from the *Perform* screen), methods for turning themselves off and other actions on, and many others. In addition, since actions are a subclass of productions (and thus collections), they may also have a list of functions, not associated with stimulus-response, which are executed any time that action is executed, as well as a repeat-count and a weight.

Action-table and Perform Screen. Actions are usually placed in the *action-table* which is defined as an HMSL collection with a few added utilities. The action-table, when turned on, is executed repeatedly by the PE, which scans all the actions as fast as possible (concurrent with all other heterarchical scheduling), and executes them accordingly. Once an action is placed in this table, it will appear on the *Perform* screen for user editing. On the *Perform* screen actions may be turned on or off with the mouse.

When an action is on (highlighted on the screen), its stimulus field will be repeatedly executed, and a flag left on the stack will be tested by the response field to determine whether a response will occur. When an action is off, it is considered to be "asleep" and will not execute its stimulus. The action-table itself may be turned on or off from the screen, turning off the *Perform* environment, or all stimulus response activity (at least concerning actions). Note that actions, unlike jobs, have no concept of time, they just "do their thing" whenever they can, depending on their stimulus and response.



EXAMPLE 5: PERFORM SCREEN

One common technique is to use probabilistic stimuli to determine the general rate of an action's execution, but there are other ways to achieve this effect as well. Each action has a local software counter, and the actions all share a global counter. These can be used for scheduling purposes, in addition to the many other types of scheduling possible in HMSL. Stimuli and responses of actions may also have access to the system's real-time clock, as well as any other external hardware timer (e.g. MIDI clocks).

The action-table and *Perform* screen have some other features. The action-table, which currently has room for sixty-four actions, is divided into four *priorities*, of sixteen spaces each. Each of these priorities has a value, and the four values taken together are used as probability ratios for the likelihood of execution. For example, the default system priorities are 5, 8, 13, 21 (a Fibonacci sequence)—actions of the first priority will tend to be executed 5/8 as often as those of the second, and 5/21 of the fourth, and so on. Actions can be placed in any one of those four priorities by the user (there are `PUT.PRIORITY:` and `GET.PRIORITY:` methods defined for actions). The default *behavior* for the action table ignores these priorities, executing all actions with an equal probability as often as possible. This is called the *unweighted* behavior. The *weighted* behavior, which, like the unweighted can be selected from the *Perform* screen, uses the priorities to stochastically decide which action to execute. Priorities may be changed on the screen or from software. Actions, once placed in the table, may be moved from one priority to another on the screen; another screen function allows for all of the actions in a given priority to be deleted.

Virtual Device Interface (VDI)

The final major conceptual module of HMSL is the *Virtual Device Interface* (VDI), designed to make HMSL as flexible as possible in communicating with artistically oriented devices, so as not to tie the system too closely to any specific sonic or other form of realization. Decisions about the specific application of data are often deferred in HMSL; for example, from shape to player, from player to instrument, and even within the instruments themselves, the main building blocks of the VDI.

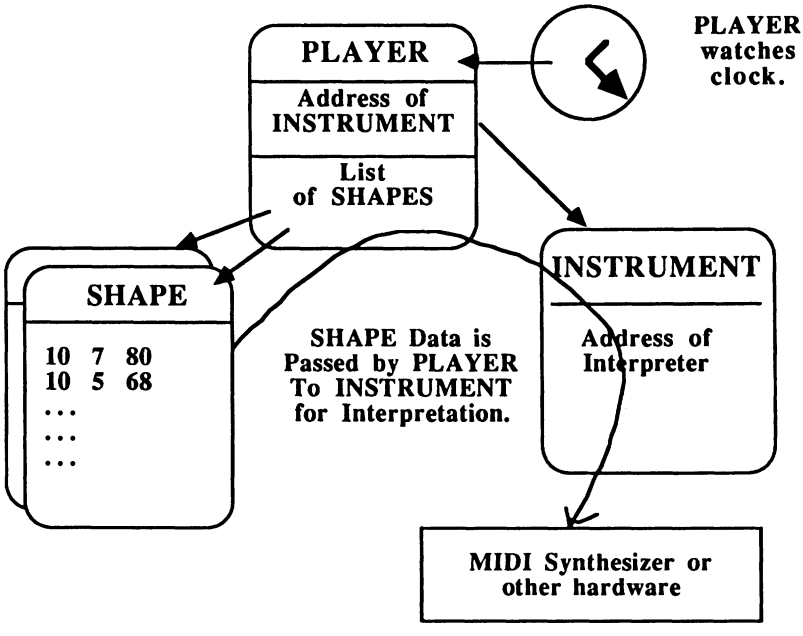
The flexible, non-hardware-dependent design of HMSL has facilitated its application to a wide variety of artistic applications, musical and otherwise. HMSL could be described as simply a highly intelligent, user-extensible, general-purpose heterarchical scheduler with a completely open-ended hardware interface; artists have found it extremely useful for controlling video, animation, sound sculpture, and other environments. In these regards, the VDI has been an important concept. By designing the hardware-specific software tasks in the definition of instruments, these nuances are kept hidden from the rest of the compositional and scheduling software, and the task of interfacing algorithmic intelligence with the often mundane but difficult job of communicating with new hardware is modularized (as in the design of UNIX drivers). The result is that the artist can spend more time thinking in creative terms about the form and organization of ideas, and less about the specifics of device interfaces.

INSTRUMENTS

The VDI mainly consists of the class of objects called *instruments*, which make use of many other subclasses as utilities. Instruments are intended to be the final software stage of most timed data in HMSL, although they can be bypassed. Instruments can be among the most complicated and sophisticated user software of HMSL; an instrument can be as intelligent as desired, and do just about anything to its input data. There are many tools for the design of instruments, as well as some commonly used default instruments that can serve as easily customizable models for the beginning user.

Instruments use special functions called *interpreters* to perform desired tasks. Interpreters take shape data and usually cause sonic output. An interpreter is passed three arguments on the stack: the current element number of the shape being executed (usually by a player, although jobs may contain instruments as well), the shape itself, and the instrument of which it is a "part." The interpreter is responsible for deciding what to do with the *n*-dimensional data at a given element number in the shape. The

interpreter may also make use of the various methods of its associated instrument, such as those pertaining to offsets, translations, allocation of "voices" and so on, since it "knows" which instrument it is in (see Example 6).



EXAMPLE 6: THE VDI: SHAPE/PLAYER/INSTRUMENT INTERACTION

HMSL includes several predefined instruments which are useful for simple applications, and serve as templates for user modification and customization. These include a class of MIDI instruments, and instruments for using Amiga local sound.[16] Instrument design has been an interesting area of experimentation in HMSL. Several users have designed MIDI system exclusive instruments for complex real-time control of parameters of various MIDI devices.[17]

One of the default instruments, MIDI instruments, illustrates many of the instrument class features. The interpreter for the MIDI instrument, if used inside an HMSL player, takes the second dimension of the player's shape(s) to be MIDI "note" ("key," "pitch"), and the third dimension to be MIDI velocity. It disregards all other dimensions, and the first dimension is assumed to be duration (this is fixed in the player). When the player sends an element number of the shape to this instrument, the values of the second and third dimension of the shape are sent out as MIDI data.

A detailed example of Late Binding in an INSTRUMENT and interpreter. The above is an example of the importance of the feature of late binding, discussed earlier in the section on ODE. Inside the definition of an interpreter, which is just a Forth word that might be used inside many instruments, methods of the instrument class will very likely be used. The interpreter can't know which instrument(s) it will be used in, so the interpreter must assume that information will be passed to it, and must have a way of using an instrument class method without knowing the specific name of the instance of that class in which it is being used. For example, one interpreter could be used in multiple instances of the class of MIDI instruments (`INS-MIDI-1`, `INS-MIDI-2`, et cetera). The interpreter must have a way of querying the object which uses it for that object's particular identity. Late binding is almost always used in this circumstance.

For example, the following simple interpreter definition takes the second dimension of a shape (which here is referenced as 1, since counting starts from 0), and sends that value as pitch to a MIDI device, added to the current offset value of its instrument, with a random MIDI velocity between 30 and 80. The value in the fourth dimension is used as MIDI program value, selecting a preset on the current MIDI device. This interpreter thus assumes the shape to be four-dimensional (consisting of duration, pitch, velocity, and program, though the actual velocity values of the shape are not used!).

```
1) VARIABLE CURRENT-INSTRUMENT
2) : MY-INTERPETER  ( element# SHAPE INSTRUMENT -- )
3)       current-instrument !  ( -- el# SHAPE )
4)        get: []                ( -- d p v pr)
5)        current-instrument @ put.preset: []
               ( -- d p v )
6)        drop  ( -- d p )
7)        current-instrument @ get.offset: []
                ( -- d p offset )
8)        +     ( -- d p+offset )
9)        80 30 wchoose midi.noteon  ( -- d )
10)       drop                  ( -- )
11) ;
```

Comments on the above code:

(1) Define a variable called `CURRENT-INSTRUMENT`.

(2) Define a Forth word called `MY-INTERPRETER`, with the conventional

interpreter stack diagram (on entry: `el#` `shape` `instrument`, with the `el#` on the bottom of the stack).

(3) Store the instrument in the variable for later use (popping it from the stack).[18]

(4) Use the `GET:` method for shapes, which retrieves the multidimensional values (as successive stack entries) at the specified index from the specified shape. Note that this is also an example of *late binding*: the interpreter has no way of knowing which shape it will be interpreting. "`d`," "`p`," "`y`," and "`pr`" stand for what these values are assumed to represent: duration, pitch, velocity, and MIDI program. With the exception of this fourth dimension for program data, this is a common HMSL format for shape data in MIDI situations.

(5) Send preset data to the current instrument. Note that `PUT.PRESET:` is a predefined HMSL method for MIDI instruments, which makes it relatively simple to change presets.

(6) `Drop` the velocity value from the stack (this interpreter will supply a random value, and ignore what is in the shape.

(7) Retrieve the instrument from the variable, placing its address on the stack. Late bind the methods `GET.OFFSET:` to that instrument, which returns a value stored in the instrument's offset instance-variable, used for switching keys, transposing, and so forth.

(8) Add that value to the value of the shape's second dimension ("pitch").

(9) Choose a random number between 80 and 30 for velocity, and send the offset pitch and that random velocity to the MIDI library command, `MIDI.NOTEON`, which has the stack diagram: (pitch vel --), and sends those values to a MIDI device. (At this point a more sophisticated method, `NOTE.ON:` could have been used, which keeps track of the values of pitch and velocity for a current instrument ,so that notes can be later turned off. I choose the simpler `MIDI.NOTEON` for this example to illustrate the use of low-level MIDI words in interpreters.)

(10) Discard the duration value in dimension 0 (it was presumably already used by the player to know when to actually call this instrument and interpreter.)

(11) End the definition.

Interpreters are a powerful tool for radically manipulating input data. Interpreters can serve as drivers for almost any hardware (e.g. video,

animation, analog synthesis, conventional score-oriented note files, et cetera). They are a way of sending data in real time to any real or virtual device, in any format. Interfacing the language to unusual hardware and reinterpreting simple data in unusual ways are two of the most interesting ways HMSL has been used.

The instrument class is quite extensive, with a wide variety of utilities. These are extensively documented in the HMSL manual and elsewhere (Polansky 1987b; Polansky, Burk, and Rosenboom 1987). Some of the tools are briefly described below.

Translators and other instrument utilities. Translators are a subclass of arrays that take in one value and return another. They may be used for translating indices into scales, or for any other type of function. Typically they convert from one numeric system to another. Examples might include converting a generic note index to a MIDI value in a specific set of pitches (i.e. gamuts in instruments), or converting a note index to a pitch or period value for an instrument with a given tuning.

Translators are used inside instruments and interpreters, and are table- or function-driven: they may look up their values or generate them. They also have a `DETRANSLATE:` method for performing an inverse function. This is only the case with table-driven translators—a `DETRANSLATE:` method would have to be written by the user for a function-driven translator. Subclasses of the translator class include the classes *tunings* and *tuning.ratios*. These facilitate experimental intonation in HMSL instrument design (Polansky 1987b). There is also a powerful class, also a subclass of arrays, called *allocators*, which is extremely useful in instrument design for deciding how many voices to "allot," and how to share these voices among several instruments.

Other HMSL Libraries

HMSL includes several libraries and utilities, in order to aid users in designing their own software. A few of these will be briefly described here.

MIDI

The MIDI library is a set of Forth words that fully implement the MIDI standard. From anywhere in HMSL one can write something like the following:

```
1 MIDI.CHANNEL!
30 MIDI.PRESET
40 56 MIDI.NOTEON
```

This will change the "current" MIDI channel to 1, change the preset on the device communicating on channel 1 to "preset 30,"[19] and send the MIDI key value of 40 and the MIDI velocity value of 56 to that device. The user has access to the MIDI toolbox at all levels of code, from high-level words such as the above, to lower-level words which send MIDI bytes and so on. HMSL also fully supports the MIDI Standard File Format, and allows for capturing pieces generated by HMSL to MIDIFiles. As stated above, MIDI output in HMSL is event-buffered, and on the Macintosh, utilizes the Apple MIDI Manager.

MIDI Input Parser. HMSL also includes a flexible utility for MIDI input, called the *MIDI Input Parser*. This feature allows users to vector any incoming MIDI data to their own routines. All of the functions in the MIDI standard are included in the HMSL MIDI parser, but none of them need be defined conventionally.

For example, a note from a MIDI keyboard can "control" the corresponding note on another keyboard, or in other words, to use the HMSL like a giant "MIDI thru," one would place the following simple routine in the appropriate MIDI input vector:

```
'C MIDI.NOTEON MP-ON-VECTOR !
```

(Comment: Place the address of the Forth word `MIDI.NOTEON`, which has been discussed above, in the MIDI parser "on vector.")

In this way HMSL echoes the note and velocity from the MIDI device connected to MIDI "in" to the device connected to MIDI "out." A slight variation on this simple example is to exchange velocity and keyboard values, so that the harder a key is hit the higher it sounds, and the "pitch" of the key struck directly corresponds to loudness:

```
: EXCHANGE.MIDI.P/V
   ( p v -- ,  MIDI note with swapped values )
       swap              ( p v -- v p )
       midi.noteon
;
'C EXCHANGE.MIDI.P/V MP-ON-VECTOR !
```

It can be seen, with a bit of imagination, that with this generalized MIDI parser, complex MIDI input data can be interpreted intelligently and complexly. Since there are vectors for every MIDI command, including all of the system-exclusive commands, MIDI becomes less a restrictive "standard" in HMSL than a general form of user-defined data exchange. The MIDI parser runs concurrently with HMSL, so that it is always parsing the MIDI input stream and executing the routines in the proper vectors. Other parts of HMSL, like actions, jobs, and productions, can change the executable addresses in the MIDI input parser vectors, so that the actual interpretation of MIDI input data can be dynamically controlled in real time in response to stimuli or algorithmic ideas. The MIDI parser itself can be turned on or off from any HMSL screen.[20]

System-Exclusive Data. Another important aspect of the HMSL MIDI environment is the ease with which *system-exclusive* data, particularly that which alters MIDI parameters in real time, is created and integrated into the system. This approach to MIDI is becoming more and more important as MIDI becomes more widespread in the computer music community and composers become more dissatisfied with the "program" notion encouraged by the MIDI concept of "note on in a given timbral configuration." This has sometimes been referred to (pejoratively, and to some extent, justifiably) as an "organ" approach. It encourages the user to first "define" an instrument that has a more or less fixed set of timbral attributes (though they may of course be enhanced and extended through the use of controllers and so on).

Even though many MIDI devices allow extremely dynamic, temporally evolving patches and programs, it is often difficult in the MIDI context to create processes that can control timbre independently of "note" events. Even the term "note," in the context of MIDI, is a semantic and protocol distinction that many composers and musicians find to be awkward at best, and pernicious at worst. HMSL, through the use of custom instrument definitions which often include system-exclusive code, allows the user to completely redefine what a MIDI event is—which may not include pitch or velocity at all.

SCORE ENTRY SYSTEM

HMSL contains a feature for entering musical data in conventional ways using note names, rhythmic values, and so on, called the *Score Entry System* (SES). The SES is a computer-keyboard-based system, and is similar to several other such environments in the ways it represents musical data. The SES runs in Forth, so that Forth, ODE, and HMSL routines may freely be

embedded in scores, and this creates further interesting possibilities for the creation of algorithmically generated pieces. Data entered in the SES is accessible to all of HMSL's data structures, and scores typed in from the computer may be played immediately as well. The SES is often used for filling shapes with melodic data.

OTHER UTILITIES

As stated above, HMSL includes a complete library for Amiga local sound and sampling. Phil Burk, in collaboration with Tom Erbe and composer Nick Didkovsky, has also successfully used HMSL to generate CSOUND score files for the Macintosh II. Some of the other important miscellaneous utilities provided in HMSL include:

(1) several standard distribution functions

(2) linear interpolator and other simple math routines

(3) complete support for real-time-clock control

(4) various and miscellaneous hardware support drivers, like those for parallel and serial ports

(5) support for various file standards, in particular the IFF and MIDIFile standards

(6) file transfer utilities for use in sharing code and libraries

(7) Motorola 56001 USP support.

The Forth compilers used for both the Macintosh (Phil Burk's *HForth*) and Amiga (Delta Research's *JForth*) versions are robust in and of themselves, supporting most of the features of the two machines. They include debugging facilities, floating-point support, command-line history, and file support. The Amiga version includes an optimizing target compiler which is quite useful in making compact executable images of pieces for performance situations.

Some Sample Applications and Pieces

This final section briefly describes some sample applications and pieces developed by composers working with HMSL. This is not intended as a comprehensive list of such work. Since HMSL has many hundreds of users, an exhaustive catalog of work done in it would be impossible at this point.

However, it is hoped that the following will show some of the experimental possibilities of the environment.

SWIRL BY PHIL BURK

Swirl exploits HMSL's ability to transform melodies in an unusual way. The source melody in this piece is considered to be a set of points in a time/pitch space. This set of points can be slowly rotated using a two-dimensional transformation matrix most commonly used in computer graphics. A rotation of 180 degrees is equivalent to a retrograde inversion. The continuous rotation can be controlled by the performer during the performance while a graphical display shows the melody at its current angle.

ADAPTIVE-TUNING MIDI APPLICATION BY PHIL BURK

Burk's adaptive-tuning software utilizes the fractional tuning capability of the Yamaha FB01, and with simple modifications to the underlying system-exclusive routines, could be easily modified for other MIDI-tunable devices. It allows a performer to play in any just-intoned or absolute tuning system using a MIDI keyboard. The system supports an "adaptive tuning," where the tuning ratio for a given interval is kept constant, irrespective of any concept of absolute pitch, much in the same way that a choir might adaptively drift by a comma or two over the course of a piece.

ROBERT MARSANYI'S UNITS AND DAVID ROSENBOOM'S "COMPOSITIONAL TOOL-KIT"

Robert Marsanyi has developed the notion of *units* in HMSL. These are similar to the standard MUSIC N idea of a unit generator. Units are arbitrarily complex software objects with graphic interfaces that allow them to be patched together on-screen like analog-synthesizer modules. These units may process or generate data in any format, including, but not limited to, sound samples and MIDI data. Units produce files in standard formats, such as IRCAM, MIDIFile, or Apple AIFF sound files. When used for synthesis or signal processing, these files may be downloaded to any one of a number of standard digital signal processors or converters. Phil Burk and Tom Erbe of the Mills CCM have developed a Macintosh II hardware and software interface for a 16-bit analog-to-digital and digital-to-analog converter (produced by Micro Technology Unlimited). HMSL, and the units in particular, can make use of this interface for playback or recording of

large sound files to and from disk. A facility for compiling unit structures into packages of code for the Motorola 56001 digital signal processor has also been implemented.

Marsanyi's units are also being used to create morphological transformation tools. MIDI input may be received in real time, processed, and played back with any of the performance tools provided by HMSL. This greatly facilitates interactive-performance works within a wide range of compositional designs.

David Rosenboom has designed a *compositional tool kit* as an extension library for HMSL. This tool kit consists of a variety of interesting generative and transformational software modules derived from Rosenboom's own musical work and generalized to have the broadest possible applicability in performance and composing. Marsanyi and Rosenboom are programming this tool kit as a set of HMSL units, which makes the items in this tool kit fully integrable with each other. All composition data generated or processed by any given UNIT can be input to any other unit in the tool kit. Tools may be freely mixed and imbedded inside parts of a compositional form. Most of these constructions are fast enough to permit them to be animated in real time. Consequently, the difference between programming for composition and programming for performance is limited.

TWO PIECES BY DAVID ROSENBOOM

Zones of Influence. Many of the above tools are kept relatively simple, but can be combined to make complex structures. Some examples of the more complex and unusual tools included in the kit are the following. Many of these are used in the piece for percussion soloist and computer music system, *Zones of Influence* (Rosenboom 1984–86).

> *Transformers*: a system for nonlinear, interactive transformation of musical parameter contours with optional classification of results according to the types of contrapuntal variations on musical lines that are produced
>
> *Evolvers*: a system for gradually transforming one musical parameter shape into another shape, including a variety of methods by which the shapes can be made to evolve with any mixture of stochastic or deterministic controls
>
> *Combinatorics*: a variety of set operations
>
> *Pattern Matcher*: a pattern-detection system, developed by Marsanyi,

for recognizing musical patterns, especially useful for sensing pattern input during live performance

Stochastics: a variety of stochastic canons optimized for live performance, including a software emulation of the classic Buchla Source of Uncertainty analog-synthesizer module

Proportional Structures: a system enabling the creation of a compositional form from the top down; a series of *bins*—definitions of time segments—can be constructed in an hierarchical arrangement (i.e. bins may be put inside other bins), the lengths or sizes of which are specified in proportional or absolute terms; HMSL-type musical structures can be stuffed inside the bins to unlimited degree; bins may be restructured by changing their position, arrangement, or size descriptions; all musical material contained in the bins is automatically rescaled to fit a new description

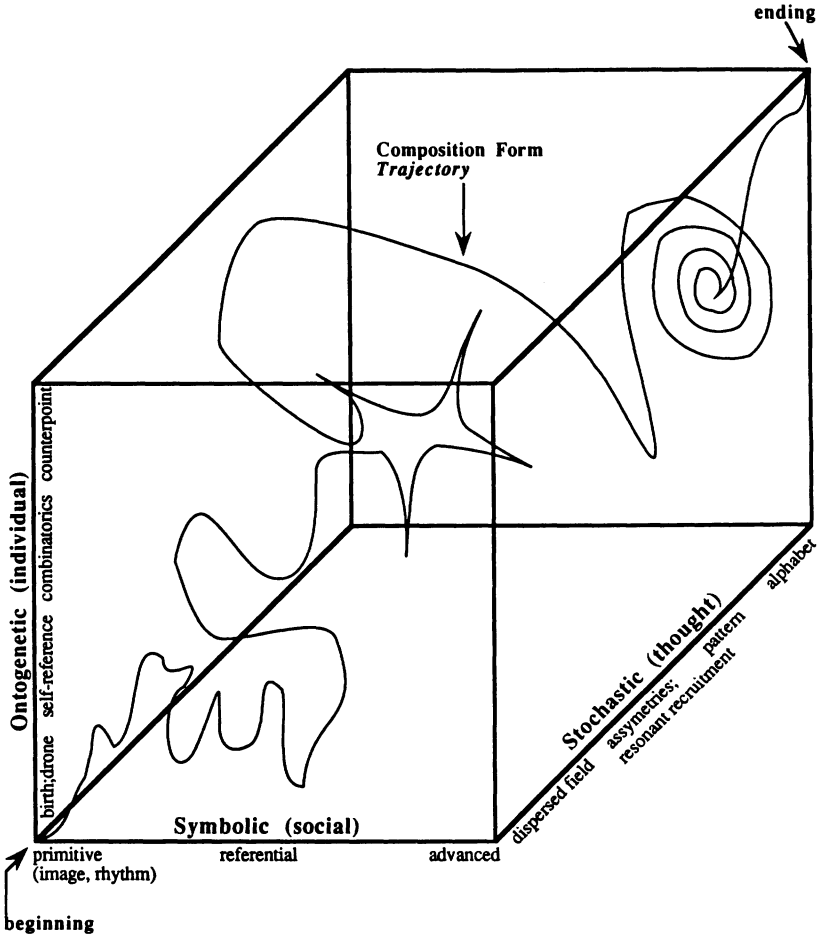
Concept Spaces: a system for mapping arbitrarily selected musical "objects" in a multidimensional scaling representation

Markov Mapper: a very fast and easy-to-use tool for specifying *n*th order sequential dependencies of musical events probabilistically using HMSL-type shapes (due to an elegant algorithm suggestion by Phil Burk).

Systems of Judgement. A recent musical work by David Rosenboom which makes use of HMSL and the notion of concept spaces is *Systems of Judgement.* This sixty-five-minute work (Rosenboom 1989) was realized with a variety of HMSL-controlled electronic as well as acoustic instruments.

A particularly interesting conceptual paradigm guided the creation of the musical form. It attempts to elucidate parallel views of evolution by examining and speculating about processes which we, any organism, or any system, must use to learn to make differentiations, be self-reflexive, and arrive at judgements from which language may be formulated.

The counterpoint of the form is conceived in a multi-dimensional concept space linking three views of evolution. The first focuses on an ontogenetic view, the evolution of the individual of a species. The second is a stochastic view of evolution by probabilistic processes. The third is symbolic of social organization. It attempts to juxtapose a scale of primitive to advanced imagery against the other two views and provide a counterpoint of semantic references that examine ideas of meaning and context. At the highest, conceptual level of the composition, these three views are mapped on the axes of a three-dimensional space. The actual music which results follows a trajectory in this space, from the origin to the upper right corner. At any given time, the music contains a mix of sound images



EXAMPLE 7: CONCEPT SPACE... FOR DAVID ROSENBOOM'S
Systems of Judgement

appropriate for the values of the coordinates referencing a particular point on the trajectory.

Subsequently, this material is realized by activating HMSL programs that have been individually constructed for various points on the three axes and integrated under a common performance-control structure. On a more concrete level of musical form specification, this approach to multidimensional scaling is being developed as a generalized tool for HMSL. The intent is for the user to be able to define axes on which selected musical parameters are scaled. Degrees of similarity among musical objects may also

correspond to their distances apart in the space under a defined metric (Polansky 1987c; Polansky, Rosenboom, and Burk 1987). Eventually, by moving objects in the space, the necessary parameters may also be updated. For the method to have great generality, however, the user must be able to arbitrarily specify the meanings of the axes defining the space, regardless of how simple or complex their subparameters may be. Consequently, a great deal of effort is going into constructing a software approach which allows any linking of multiple variables from arbitrary points in an HMSL hierarchy in order to specify how values for a given axis are generated. These correspond to what are termed the *order parameters* of a phase space in dynamic systems theory.

Rosenboom is currently working on interfacing HMSL with EEG sensing equipment to continue much of the research he did in biofeedback and the arts during the 1960s and 1970s (Rosenboom 1976, 1990). The revival of this work is now focused on production of a large-scale, self-organizing opera, *On Being Invisible II*. This work, which is based on Rosenboom's mid-seventies biofeedback piece, *On Being Invisible* (Rosenboom 1977a, 1977b, 1984), involves an extensive, real-time algorithmic composition system. Elements of the system include: a principle component analysis of auditory event-related potentials from the EEGs of performers; a model of musical perception, based on parametric-difference detection, pattern analysis, and a memory-persistence algorithm; and a hierarchical musical-structure builder that constructs a complex musical form based on shifts of musical attention detected in various performers. All of the musical form building and performance aspects of this project are being programmed in HMSL. HMSL's stimulus-response environment is being employed to design networks in which musical processes are affected by neurological and biological functions. This work is described in Rosenboom 1990.

Notes

1. An important concern was "input structures"—the ability of the system to respond to a wide variety of kinesthetic, software-generated, and electronically generated stimuli. This is a particularly interesting aspect of some of the electronic instruments built by Rosenboom and Donald Buchla, most notably the TOUCHE, a keyboard-based instrument that was extremely flexible in its response to digital and analog input. The pioneering work of Rosenboom, especially in his music with brainwave and other biofeedback control mechanisms, and Donald Buchla (and his associate, programmer Lynx Crowe), was a starting point for the design of the more general, completely software-based stimulus-response features of HMSL.

2. Some of the composers who had used Forth extensively in the 1970s include Jim Horton, John Bischoff, Tim Perkis, and Rich Gold (these four made up the League of Automatic Composers in the San Francisco Bay Area), George Lewis, Bill Maginnis, Joel Ryan, Donald Buchla (Lynx Crowe's MIDAS language for the Buchla 400 digital-keyboard synthesizer is written in Forth), Martin Bartlett (who has also written a music language for the Buchla 400, called MABEL), Ron Kuivila, David Behrman, and many others.

3. For the past ten years, the Mills CCM has hosted a series of guest lectures called the Seminar in Formal Methods, frequently about live interactive computer-music systems (Polansky and Levin 1987). The ideas presented in that series have inspired various aspects of HMSL, and the language has benefitted from the community of artists and thinkers who have visited and worked at the CCM in recent years.

4. Nearly simultaneously with HMSL, two other Forth-based small computer real-time computer-music languages have evolved. They are well worth noting here, because of their importance and because of the similarities and differences in design philosophy with HMSL. David Anderson and Ron Kuivila's FORMULA, first written for the Apple II and later ported to the Atari 1040ST and the Macintosh, is a "process-oriented" language, with sophisticated and accurate scheduling. Like HMSL, it allows the user to write his or her own music code. FORMULA's emphasis seems to be an elegant economy of words (it exists as a small superset of Forth), and an extremely accurate multi-process scheduler, which reflects its authors' theoretical ideas in this regard (Anderson and Kuivila 1986a, 1986b, 1989). FORMULA's scheduling ideas have influenced those of HMSL somewhat (in fact Ron Kuivila worked a little on the prototype of HMSL

at the CCM). HMSL's MIDI event-buffering system is in part inspired by FORMULA, but the greater complexity and diversity of system data structures in HMSL, each with its own scheduling "intelligence," has made general event-buffering a more difficult task of HMSL than for FORMULA, where there is fundamentally only one type of musical event data structure, the "process" (Anderson and Kuivila 1988), albeit a compact and tremendously flexible one.

MASC, developed by Daniel Kelley and Allen Strange at San Jose State University, is a small, extremely portable, and very useful Forth-based language. MASC has the ability to easily specify arbitrary envelopes for any parameter, from timbral to formal. MASC was originally written as a hybrid-control language, and later adapted to a MIDI environment. MASC also allows for user-written FORTH code. Since it is written in public domain Forths for each of its implementations (unlike HMSL), and is itself public domain, it is free, and easy to port to almost any small system (in contrast to HMSL and FORMULA, each of which exist on only two machines, and are more difficult to port). Its data structures are also less extensive than HMSL's (and thus, to its credit, much easier to learn). Like FORMULA, MASC offers the user a standard Forth environment as a programming language. For an interesting overview of these languages, and several others, see Scholz 1988.

These languages, together with HMSL, are evidence of the tremendous interest in music language design for small computers that started around 1980. It is interesting also, that none of these implementations adopted the philosophy of "application" or "program"—they all assumed the user's ability to program in Forth. This philosophy has unfortunately been almost completely disregarded—to the detriment of a certain type of musical experimentation—by the many music software companies that have sprung up following the wide acceptance of MIDI.

5. Other programming environments can be used simultaneously within HMSL. All versions of Forth used in the various implementations include full-featured 68000 Assembler environments, which are often useful for user-designed custom hardware control, speed-critical applications, and so on. The Amiga version supports the calling of C structures from HMSL, particularly for use of the machine's libraries, and the Macintosh version has a similar facility. Users have implemented other high-level languages, like a version of LISP (by New York composer Nick Didkovsky), within HMSL. A 56001 assembler is currently being integrated into certain HMSL environments.

6. We have begun to think, rather informally, that the word "hierarchical" is a slight misnomer. HMSL's data structures are all indeed holarchical, or more precisely, *heterarchical*, meaning that most data types can be contained in, or can contain almost any other (for a related interpretation of these terms, see Abraham 1986 and 1987). The word hierarchy suggests, to some extent, the notion of "authority," or more applicably here, a notion of rigidity of organization of data levels, implying that some data-types are "larger" and more "powerful" than others in a formal sense. The choice of the word "hierarchical" reflects the authors' current theoretical thinking at the time HMSL was designed, some ten years ago, and was partially derived from the theoretical ideas of James Tenney (Tenney 1987; Tenney and Polansky 1980). However, Tenney himself has revised his thinking, and has stated that he would like to replace all occurrences of the work hierarchical with holarchical in his earlier work.

7. Examples of this are Polansky's class of Metric-Shapes, which contain methods for morphological metrics (distance functions) between themselves, and which are used in the piece *17 Simple Melodies of the Same Length* (Polansky 1988), and Henry Lowengard's `SCRAMBLE:` method. This was originally an unimplemented method on the *Shape-Editor* screen. Lowengard wrote his own, which then became integrated, after some modifications by Burk, in the canonical version of the system. This is also a good example of the way that users have contributed to the development of the language itself.

8. There is one notable exception to this statement about "heterarchy of any complexity," which is that no morph may contain itself anywhere in its "lineage," either as a child, grandchild, great-grandchild, or any generation of uncle, aunt, nephew, and so on. In other words, no heterarchical morph may execute itself while it is currently executing itself. Obviously, from a programming standpoint, this would invoke a kind of infinite loop. From a musical standpoint, trying to find a way to include recursion, while philosophically intriguing, seemed unnecessary, since morphs can be created and rearranged by the system easily (with dynamic instantiation, and by the various intelligent morphs). In the current version of HMSL, if a morph is executed while it is already being executed (that is, if it is already in what is called the *active object list*), the first occurrence of the morph will be stopped, and the new occurrence will start from the beginning, like plucking a string while it is still vibrating. This procedure, incidentally, relieves the user from the burden of worrying about whether a morph ever includes itself, for if it does (perhaps by

inserting itself in response to some virtual or physical stimulus), it will simply abort the execution of the higher-level occurrence, and start over again. This means that in fact, morphs can be "arranged" recursively, but that no real recursive execution takes place.

9. An example of this is in some of the work of Steven Miller, a recent graduate student in electronic music at Mills, who has made excellent use of structures in creating long, "ambient" music environment installations. Miller's structures, in a recent piece called *Motion/Stasis* make decisions based on a combined measure of the weight and link tendencies of component morphs, and choose new behaviors for the structure itself based upon these values.

10. "The productions [in a phrase-structure grammar] are grammatical rules that specify how sentences in the language can be made up.... A production specifies that string α can be transformed into string β" (Liu 1985, 53–54). In HMSL, we originally considered shapes to be a kind of analog to linguistic strings, and that our productions would transform shapes and combine them into larger morphological units ("sentences").

11. This definition of duration is not restrictive in HMSL, the composer may define the concept in any number of ways depending on how he or she wishes to describe timed musical events. For example, the *on-times* of events may be longer than their *durations*, resulting in simple polyphony. The definition given here is most useful in describing the way that simple, canonical versions of players schedule "next events," but it is easy to extend, both conceptually and practically within HMSL.

12. Flurry (1988) points out that in object-oriented programming the term "polymorphism" has a more general meaning: that the same method sent to objects of different classes has different results. Although this sense of the term does not apply to the way we refer to the HMSL scheduler, it nevertheless accurately describes the way that various types of MORPHS are scheduled: most of them have the same methods, like `EXECUTE:` and `DONE:`, which take very different forms for different MORPHS.

13. The term "epochal" is borrowed from Tenney (1987, 106): "Epoch refers to the moment of occurrence—in the ongoing flow of experienced time—of any musical 'event,' compared to some reference moment such as the beginning of the piece."

14. The shape in this illustration is taken from Polansky's *Cocks Crow, Dogs Bark, This All Men Know, but Even the Wisest Cannot Tell Why*

Cocks Crow, Dogs Bark, When They Do (Polansky 1988) for several interacting computers and voice (which controls and is processed by the computer network). This piece, a collaboration with composer John Bischoff and poet Melody Sumner, will be described in a future article on Polansky's work. This particular dimension of this shape controlled the harmonic complexity of a four-part chord by stochastically selecting higher and higher prime partials depending on values in that dimension.

15. As in the choice of the term "production," the term "action" has its genesis in the fields of artificial intelligence and cognitive psychology (e.g. McCulloch and Pitts 1943), but has been modified for use in HMSL.

16. The Amiga has four 8-bit DMA sound channels, all under software control in real time. Although it is not high-fidelity sound, this is a flexible tool for experimentation in the areas of sampling, synthesis, and intonation. HMSL includes support for these local sound libraries. For more information on the Amiga internal sound, see the Amiga manuals themselves, or any one of a number of articles written in the various trade magazines.

17. An example is Polansky's real-time system-exclusive implementations for the Roland DEP-5 and Yamaha FBO1, which have been used in several live works such as *Cocks Crow, Dogs Bark....* and *Simple Actions* (Polansky 1988). Another example is a system-exclusive implementation for the Roland S-50 digital keyboard instrument, created by Jeanne Parson, a graduate student in electronic music at Mills College. Parson's code allows for powerful real-time parameter modification of that instrument that would not normally be possible. Parson used this code extensively in a work for violin and electronics titled *Levels of Complexity.*

18. Forth programmers will note that this is not particularly elegant Forth code. The use of local variables, or a few simple `SWAP`s and `OVER`s (stack manipulation words) would be more conventional Forth style, but global variables and a more straightforward style keep this example conceptually simple.

19. "Preset" is synonymous with "program" in MIDI, but there is also a command called `MIDI.PROGRAM`, which, illustratively enough, is defined as:

```
: MIDI.PROGRAM MIDI.PRESET ;
```

This shows one of the advantages of the Forth environment for music—anything can be renamed, or aliased, trivially, and the user has rapid and extraordinary control over the taxonomy and nomenclature of musical ideas. If one does not like the terms "collection," "shape," "structure," "production," it is trivial (and in fact, to some extent, encouraged by us) to customize these terms to, for example, Ward, June, Wally, and The Beave.

20. One aspect of the design philosophy in HMSL is to make everything as "open-ended" as possible, providing an environment for the widest possible range of experimentation. The MIDI parser is a good example of this: the restriction of a one-to-one software correspondence between the MIDI standard and what actually "happens" is avoided entirely, and MIDI becomes, in some sense, just a nonsoftware defined, inexpensive hardware interface. This does not mean that one does not adhere to MIDI conventions, but that one can extend them as far as desired. The facility for generalized interpretation of input data is of paramount importance in the design of real-time computer-music languages.

References

Abraham, Ralph H. 1987. "Mathematics and Evolution: A Proposal." *International Synergy Journal* no. 5:27–45.

———. 1986. "Mathematics and Evolution: A Manifesto." *International Synergy Journal* no. 3:14–27.

Anderson, David, and Ron Kuivila. 1989. "Continuous Abstractions for Discrete Event Languages." *Computer Music Journal* 13, no. 3, (Fall): 11–13.

———. 1988. *The FORMULA Reference Manual.* Available from the authors.

———. 1986a. "Accurately Timed Generation of Discrete Musical Events." *Computer Music Journal* 10, no. 3 (Fall): 49–56.

———. 1986b. "A Model of Real-Time Computation for Computer Music." In *Proceedings of the 1986 International Computer Music Conference*, edited by Paul Berg, 35–41. San Francisco, CA: Computer Music Association.

Burk, Phil. 1987. "HMSL—An Object Oriented Music Language." *RoboCity News* 3, no. 4 (May):6–8.

Burk, Phil, M. Haas, B. Donovan, and J. King. 1989. *JForth Professional for the Amiga: User Manual and Reference Guide, Version 2.0.* San Rafael: Delta Research Incorporated.

Burk, Phil, and Polansky, Larry, (with R. Marsanyi, D. Hayes, and M. Gass). 1990. *HMSL (Hierarchical Music Specification Language), Reference and User Manual, VERSION 4.0.* Oakland: Frog Peak Music.

Cox, Brad J. 1986. *Object Oriented Programming: An Evolutionary Approach.* Reading, Massachusetts: Addison-Wesley.

Flurry, Henry S. 1988. "The CREATION STATION: An Approach to a Multimedia Workstation." *Leonardo: The International Journal of the Arts, Sciences and Technology,* supplemental issue, "Electronic Art": 31–38.

Goldberg, Adele, and David Robson. 1983. *SMALLTALK-80: The Language and Its Implementations.* Reading, Massachusetts: Addison-Wesley Publishing.

Kelley, Daniel, and Allen Strange. *MASC Reference Manual.* Available from the authors c/o Electro-Acoustic Music Studio, San Jose State University, 1 Washington Square, San Jose, California 95192.

Liu, C.L. 1985. *Elements of Discrete Mathematics.* New York: McGraw-Hill.

Lohner, Henning. 1986. "The UPIC System: A Users' Report." *Computer Music Journal* 10, no. 4 (Winter): 42–49.

McCulloch, Warren S., and Walter Pitts. 1943. "A Logical Calculus of the Ideas Immanent in Nervous Activity." *Bulletin of Mathematical Biophysics* 5:115–33.

Polansky, Larry. 1987a. "Distance Music I–VI, For Any Number of Programmer/Performers and Live, Programmable Computer Music Systems." *Perspectives of New Music* 25, nos. 1 and 2 (Winter and Summer): 537–44. (Includes score for *Simple Actions.*)

———. 1987b. "The HMSL Experimental Intonation Environment." *1/1: The Journal of the Just Intonation Network* 3, no. 1 (Winter): 4–15.

———. 1987c. "Morphological Metrics: An Introduction to a Theory of Formal Distances." In *Proceedings of the 1987 International Computer Music Conference*, compiled by James Beauchamp. San Francisco, CA: Computer Music Association.

———. 1988. *Works for Performers and Live Interactive Computer.* Oakland: Frog Peak Music Cassette. (Contains recordings of four works done using HMSL: *B'rey'sheet*; *Simple Actions*; *Cocks Crow, Dogs Bark...*; and *17 Simple Melodies of the Same Length.*)

Polansky, Larry, and J. Levin. 1987. "The Mills College SEMINAR IN FORMAL METHODS Series: A Documentary History of the First Five Years." *Leonardo: The Journal of the International Society for the Arts, Sciences, and Technology* 20, no. 2:155–63.

Polansky, Larry, and David Rosenboom. 1985. "HMSL (Hierarchical Music Specification Language), A Real-Time Environment for Formal, Perceptual and Compositional Experimentation." In *Proceedings of the 1985 International Computer Music Conference*, edited by Barry Truax, 243–50. San Francisco, CA: Computer Music Association.

Polansky, Larry, David Rosenboom, and Phil Burk. 1987. "HMSL: Overview (Version 3.1) and Notes on Intelligent Instrument Design." In *Proceedings of the 1987 International Computer Music Conference*, compiled by James Beauchamp. San Francisco, CA: Computer Music Association.

Pope, Stephen Travis. 1986. "Music Notation and the Representation of Musical Structure and Knowledge." *Perspectives of New Music* 24, no. 2 (Spring-Summer): 156–89.

———. 1987. "A Smalltalk-80-based Music Toolkit." In *Proceedings of the*

International Computer Music Conference, compiled by James Beauchamp, 166–73. San Francisco, CA: Computer Music Association.

———. 1989. "Machine Tongues XI: Object-Oriented Software Design." *Computer Music Journal* 13, no. 2, (Summer): 9–22.

Rosenboom, David. 1976. *Biofeedback and the Arts: Results of Early Experiments*. Vancouver: ARC Publications.

———. 1977a. *On Being Invisible*. LP Record, Music Gallery Editions (Toronto) MGE-4.

——— 1977b. *On Being Invisible*. Video tape. Western Front Video, Vancouver.

———. 1984. "On Being Invisible." *MusicWorks* 28 (Summer): 10–13.

———1984–86. *Zones of Influence*. Score for percussion soloist, computer music instrument, and auxiliary melodic parts. Oakland: Frog Peak Music.

———. 1987a. *Collected Articles*. Oakland: Frog Peak Music.

———. 1987b. "Cognitive Modelling and Musical Composition in the Twentieth-Century: A Prolegomenon." *Perspectives of New Music* 25, nos. 1 and 2: 439–46.

———. 1987c. "A Program for the Development of Performance-oriented Electronic Music Instrumentation in the Coming Decades: 'What You Conceive Is What You Get'." *Perspectives of New Music* 25, nos. 1 and 2: 569–83.

———. 1989. *Systems of Judgement*. Compact disk of work for computer and acoustic music instruments. CDCM Computer Music Series, vol. 4. Baton Rouge: Centaur Records.

———. 1990. *Extended Musical Interface with the Human Nervous System: Assessment and Prospectus*. Leonardo Monograph Series No. 1. Berkeley, CA: ISAST International Society for the Arts, Sciences, and Technology.

Scaletti, Carla. 1987. "KYMA: An Object-Oriented Language for Musical Composition." In *Proceedings of the 1987 International Computer Music Conference*, compiled by James Beauchamp, 49–56. San Francisco, CA: Computer Music Assocation.

———. 1989. "Composing Sound Objects in KYMA." *Perspectives of New Music* 27, no. 1 (Winter): 42–69.

———. 1989a. "The Kyma/Platypus Computer Music Workstation," *Computer Music Journal* 13, no. 2, (Summer): 23–38.

Scholz, Carter. 1988. "MIDI Resources." *Keyboard* (November): 74–88.

———. 1988a. "HMSL Software Language." *Music Technology*, (September): 82–83.

Tenney, James. 1987. *META + HODOS and META Meta + Hodos* (written in 1961). Oakland, CA: Frog Peak Music.

Tenney, James, with Larry Polansky. 1980. "Temporal Gestalt Perception in Music." *Journal of Music Theory* 24:205–41.