An Interactive Environment for Object-oriented Music Composition and Sound Synthesis

C.A. Scaletti *
CERL Music Project
University of Illinois at Urbana-Champaign

R. E. Johnson
Department of Computer Science
University of Illinois at Urbana-Champaign

Abstract

Kyma is an object-oriented environment for music composition written in Smalltalk-80, which, in conjunction with a microprogrammable digital signal processor called the Platypus, provides the composer with a means for creating and manipulating Sound objects graphically with real-time sonic feedback via software synthesis. Kyma draws no distinctions between the materials and the structure of a composition; both are Sound objects. When a Sound object receives a message to play, it transforms itself into a microSound object, i.e. an object representation of itself in the microcode of the Platypus. Thus an object paradigm is used not only in the representation of Sound objects in Smalltalk-80 but also in the microcode representation of those Sound objects on the Platypus.

---

*The author's current address is: Kymatics, P.O. Box 2530, Station A, Champaign, IL 61820; telephone: (217) 328-6645.

1. Sound Objects

1.1 The Problem of Sound Synthesis

Sound can be described digitally as a stream of instantaneous amplitude values called samples. A digital-to-analog converter translates this stream of numbers into a continuously varying voltage which, if used to drive a loudspeaker, is translated into a continuous variation in air pressure — sound. Recording or synthesizing frequencies of up to 10 kHz necessitates a sampling rate of 20 kHz. At this sample rate, 1 million 16-bit samples are required to represent less than a minute of sound; for stereo sound the requisite number of samples doubles. The considerable quantity of data required for digital sound synthesis presents problems with regard to both speed and manageability.

1.1.1 Speed

Much of the early work in digital sound synthesis was based on Max Mathews' "acoustic compilers", Music1 - Music5 [Loy85]. These "Music N" languages generate sound by means of *software synthesis*, that is, the stream of samples representing the waveform of the desired sound is specified exclusively in software. While software synthesis is an extremely flexible technique, most of the Music N languages were designed to run on general purpose computers in a noninteractive way; turnaround times measured in hours or even days can be frustrating for composers trying to experiment with new sounds.

One way around this problem is to implement the sound synthesis algorithms in hardware, a technique employed in the ubiquitous MIDI synthesizers. These synthesizers provide composers and performers with digitally synthesized sound in real time. A price is paid, however, in terms of flexibility; a hardwired algorithm is extremely

difficult to modify, limiting the capabilities of each synthesizer to its fixed set of algorithms.

More recently, Digital Signal Processors (DSPs) have been used to solve the speed problem in digital sound synthesis. Since DSPs are programmable and are designed specifically for samples generation, they can do software synthesis in *real time*, allowing composers to explore new sounds interactively.

1.1.2 Manageability

The Music N languages provide composers with a familiar model to assist them in specifying the sample stream — the model of a "score" performed on an "instrument". A Music N "instrument" is designed by connecting unit-generators, software modules which simulate familiar analog circuits such as oscillators or filters. In a totally separate activity, a Music N "score" is specified as a sequence of instructions for turning instruments on or off at prescribed times. (Most of the languages for controlling MIDI synthesizers also employ this model of instrument and score with obvious popular success.)

There is, however, a disadvantage to using this model; while these languages make it easy to do a "middle-level" organization at the granularity of notes, it can be awkward, in some cases impossible, to to use these same languages to organize higher level structures (e.g. phrases or sections) or lower level structures (since the "timbre" of a sound is dealt with separately as "instrument design"). What these instrument/score languages seem to lack is an abstract structure which can be applied uniformly at *all* levels of organization.

It is not possible to individually compose each of the millions of samples comprising a piece of music, nor do individual samples mean much in isolation. Some means is needed by which to organize these samples, group them together into meaningful chunks, enclose all the details in a package, give it a name, and refer to it thereafter as a single entity. In Kyma such an entity is referred to as a Sound object. Everything in Kyma, from a single timbre to the structure of an entire composition, is a Sound object. These Sound objects can be manipulated, transformed and combined into new Sound objects. Furthermore, Sound objects can be continuously redefined as work on a composition progresses; objects which were once encapsulated in other objects can be brought to the top level, top level objects can be combined and hidden within a yet higher level object.

Other music languages that acknowledge this need for a uniform structural entity include FORMES with its "process" [Rodet84] [Cointe87], HMSL with its "morph" [Polansky85,87], the SSSP project with its "musical event" [Buxton85], and Herbert Brun's language, Sawdust, with its "link" [Grossman87].

1.2 Sound objects vs. Standard Music Notation

Why adopt this strange idea of a Sound object when music already comes equipped with traditional structures specified in terms of staves, measures and notes? In order to answer this question, it is first necessary to differentiate between the use of the word *music* in reference to an acoustic event and *music* in reference to written notation. Music notation is a highly contextual list of shorthand instructions to a performer; it does *not* fully specify the acoustic event. Evidence of this can be seen in the fact that trained musicians will perform a score with the name "Bach" at the top quite differently from their performance of a score bearing the name "Chopin".

Composition environments based on music notation, while they do provide a structure for the composer, do not provide a flexible and redefinable structure. Furthermore, they tend to limit the specification of sound structures to those which could be performed on traditional instruments by human performers. This sort of self-imposed limitation is analogous to the long takes in front of a static camera typical of early films — they were using the camera to passively record the performance of a play, not yet realizing the full potential of film as a new medium distinct from the medium of live theater.

Sound objects form a superset of the set of notes, i.e. there are Sound objects which can be described in Kyma which could not be expressed using standard music notation. These are not just bizarre sound effects or outlandish examples (e.g. it would be difficult to record ordinary speech using musical notation) but include examples which fall squarely in the domain of traditional music. For example, musicians can find it difficult to notate even their *own* improvizations.

1.3 Kyma Definition

1.3.1 Definition of Sound

A Sound in Kyma is either a Sound Atom or a Transform of one or several subSounds.

A Sound is defined as:

i) SoundAtom
ii) UnaryTransform, T(s) where s is a Sound
iii) N-ary Transform, $T(s_1, s_2, ..., s_n)$ where $s_1, s_2, ..., s_n$ are Sounds

A transform is the result of applying a function to its subSounds. In this sense, a transform is something like a

prism: when viewing physical objects through a prism, the structure of the prism is made apparent by the manner in which it distorts the appearance of the viewed objects; however the prism doesn't actually alter the viewed objects, and the prism is an object distinct from the viewed objects.

1.3.2 Currently Implemented Sound Classes

Figure 1 shows the hierarchy of all Sound classes currently implemented in Kyma. (It should be emphasized, however, that Kyma is an open system, and that the composer can add to and modify this class hierarchy.)

1.3.3 N-ary Transforms

Each N-ary Transform has an instance variable, subSounds, which contains an OrderedCollection of Sounds.

Mixer and Concatenation are the primary temporal operators of Kyma. The subSounds of a Mixer are played in parallel; those of a Concatenation are played serially. A Mixer is defined to be the sum of its subSounds, a Multiplier their product, and a Concatenation as a sequence of its subSounds.

1.3.4 Unary Transforms

Each UnaryTransform has an instance variable, subSound, containing a single Sound to be transformed. For example, a Delayed is defined as a Sound which waits a specified amount of time before playing.

1.3.5 Sound Atoms

A SoundAtom has no subSounds, and, as could be inferred from its name, cannot be broken down into constituent parts. For instance, a LiveSound is defined as the input from the analog-to-digital converter; if a microphone is connected to the input, the LiveSound's samples will come from that microphone.

In a LookupFunction, a periodic signal is obtained by retrieving samples from a lookup table. A single cycle of the desired waveform is precomputed and stored in the table; by incrementing an index into this table modulo the length of the table, this cycle can be repeated any number of times. Different frequencies, i.e. numbers of cycles per second, are achieved by using different sized increments for stepping through the table.

1.3.6 Potential Sounds

A Potential Sound is one which does not respond directly to the **play** message; instead when such a Sound receives a message to play, it creates a new Sound and then sends *it* the **play** message. For example, when a Palindrome gets a **play** message, it first creates a Concatenation of its subSound and the Reverse of its subSound, and it then tells the *Concatenation* to play.

- Object
  - Sound
    - SoundAtom
      - AlgorithmicSound
        - ConstantSound
        - PeriodicSound
          - LookupFunction
            - OneShot
      - LiveSound
      - PotentialAtom
        - LiveSoundRecording
        - Pluck
        - TwoFormantVoiceElement
    - SoundTransform
      - NaryTransform
        - Concatenation
        - Mixer
        - Multiplier
        - PotentialNaryTransform
          - AMSound
          - CellularAutomaton
          - DelayingMixer
          - EndTogetherMixer
          - IMSSound
          - Playback
          - ProbabilisticSound
      - UnaryTransform
        - AmplitudeScaled
        - Delayed
        - DualParallelTwoPoleFilter
        - LinearAmplitudeEnvelope
        - PotentialUnaryTransform
          - Enveloped
          - Palindrome
          - RecursiveSplit
          - Repetition
          - TwoFormantElement
        - ScaledUp
        - SecondOrderFilter
        - SoundLevelTransform
          - DynamicTransform
            - DurationTransform
            - FrequencyTransform
          - OrderTransform
            - Retrograde
            - Reverse
        - UnaryLookup
          - Echo
          - Granulation
          - KSFilter
          - RecordForDisplay
          - Recording
          - UnaryCycle
            - FMSound
            - Panner
          - WaveShaper

Figure 1. The hierarchy of Sound classes as currently implemented in Kyma. Where one class is the subclass of another which occurs before it in the list, this is indicated by indentation.

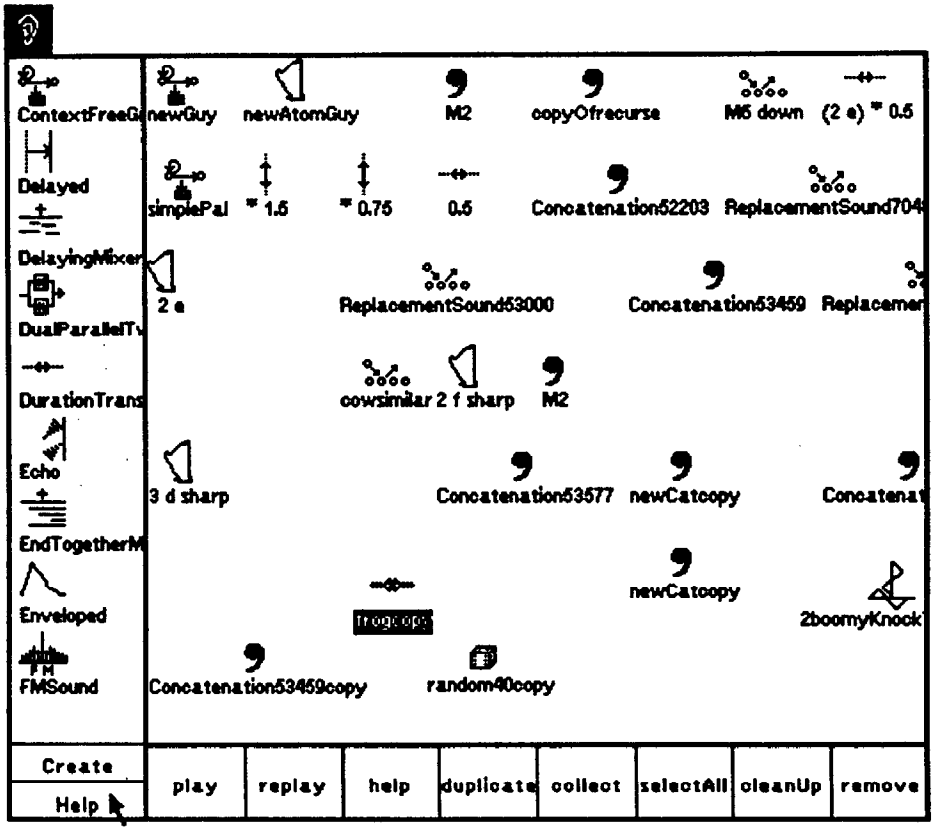

Figure 2. The Main Kyma View. Icons representing Sound classes appear in the list on the left, icons representing Sound instances (called SoundPoints) appear in a SoundPlane (the large area to the right of the class list).

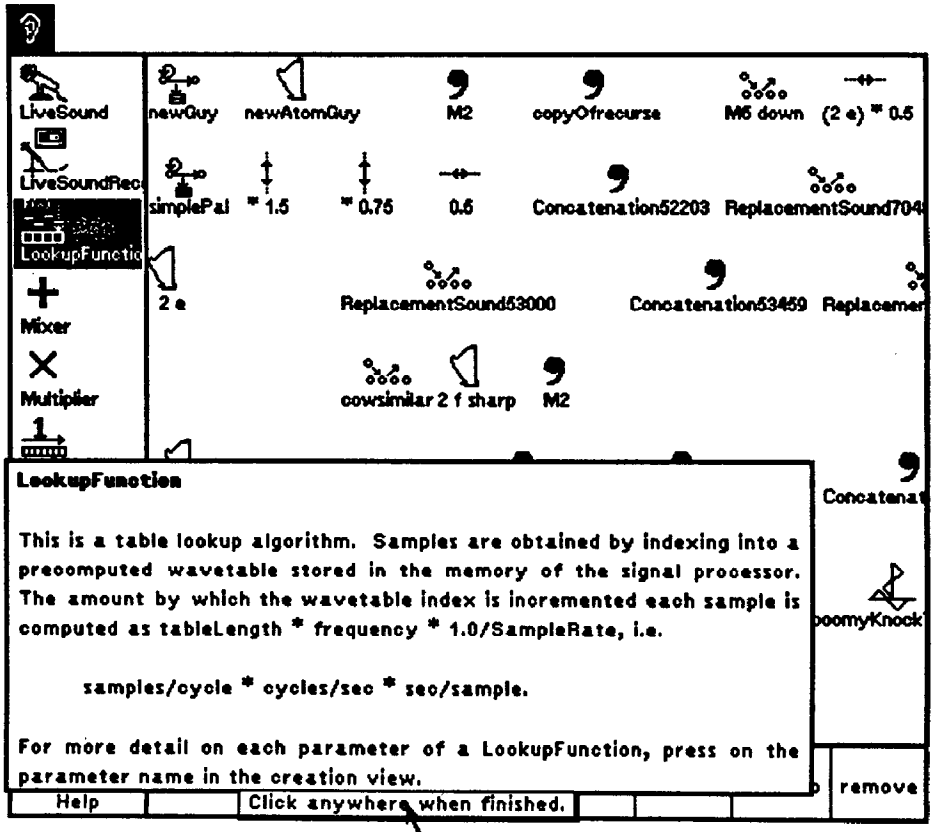

Figure 3. By pressing the Help button below the list of classes, the composer can see a brief description of the selected class.

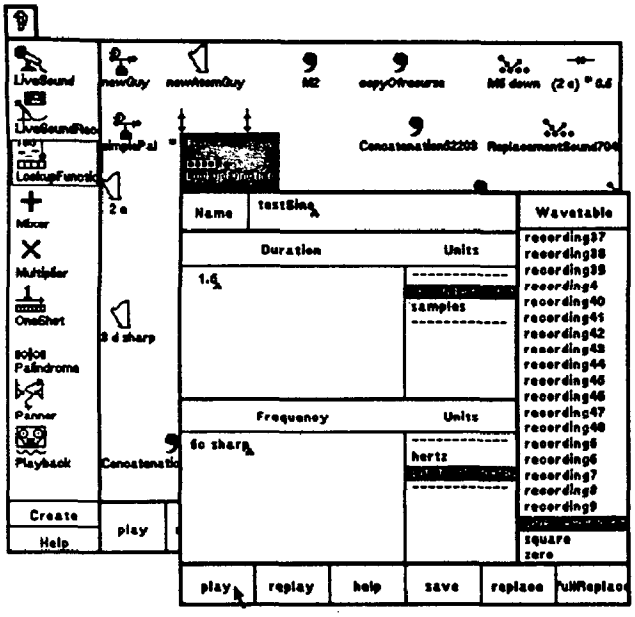

Figure 4. The Creation View for a LookupFunction. Parameters can be typed in or supplied by dragging another SoundPoint into the parameter field.

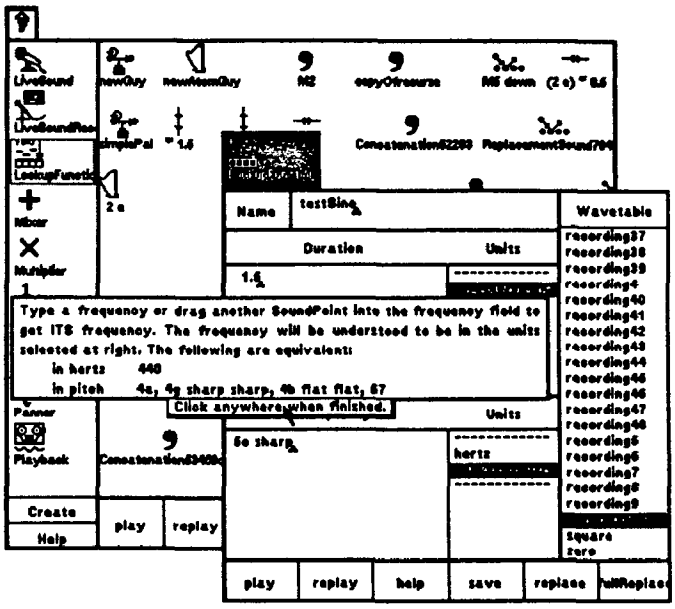

Figure 5. Pressing on a parameter name in a creation view provides a brief description of that parameter on the display.

A FrequencyTransform is an interesting potential Sound which can have a time-varying effect on its subSound. Each FrequencyTransform contains, as an instance variable, a function of time, frequency, and the duration of its subSound. (See [Hebel88] for a detailed description of Functions as ArithmeticObjects in Smalltalk-80.) Should the FrequencyTransform have an N-aryTransform as its subSound, this function will be reevaluated at each of the constituent Sounds' startTimes; it is a time-varying transformation down to the granularity of individual *Sounds*, not individual *samples*.

1.4 Using Kyma

Kyma's user interface can be characterized as a direct manipulation system. The composer can create new sound instances, group them into collections, extract parameter information from them, and examine/change their structure by clicking or dragging with the mouse.

1.4.1 The Main Kyma View

By selecting KYMA from the list of options in Smalltalk's background menu, the composer obtains a main Kyma view like that shown in Figure 2. The icons (called SoundPoints) represent Sound objects; SoundPoints can be selected, dragged, or grouped together into SoundCollectionPoints. (The appearance of these operations was modelled after that of the Macintosh Finder).

The strip along the left edge of the main Kyma view is a list of Sound classes. Pressing the help button opens a short description of the currently selected Sound class (Figure 3). Pressing the create button opens a creation view such as that shown in Figure 4, the creation view for a LookupFunction.

1.4.2 Creation Views

The composer can press any parameter name in the creation view for assistance on supplying an appropriate parameter value. Figure 5, for example, illustrates the result of pressing "Frequency" in the LookupFunction's creation view.

Each parameter can be supplied either by typing it in from the keyboard or by dragging another SoundPoint into the appropriately named pane (as in the sequence shown in Figure 6).

The creation view for Echo (Figure 7) illustrates several different ways for a composer to enter the Sound's parameters: the subSound is supplied by dragging a SoundPoint into the small SoundPlane, the name is supplied by typing or dragging another SoundPoint into the name pane, the delay time is specified by typing or dragging into the delay pane and then selecting the appropriate units from the list to the right of that pane, and the feedback factor is supplied by an adjustable slider. A particular delay line is selected from the list of all delay lines on the right.

1.4.3 Opening SoundPoints

Any of the SoundPoints or SoundCollectionPoints can be examined or altered by double-clicking their icons; this "opens up" the SoundPoint or SoundCollection to reveal its structure. The process of opening up a SoundPoint is essentially the reversal of the process used to create that

I.

II.

III.

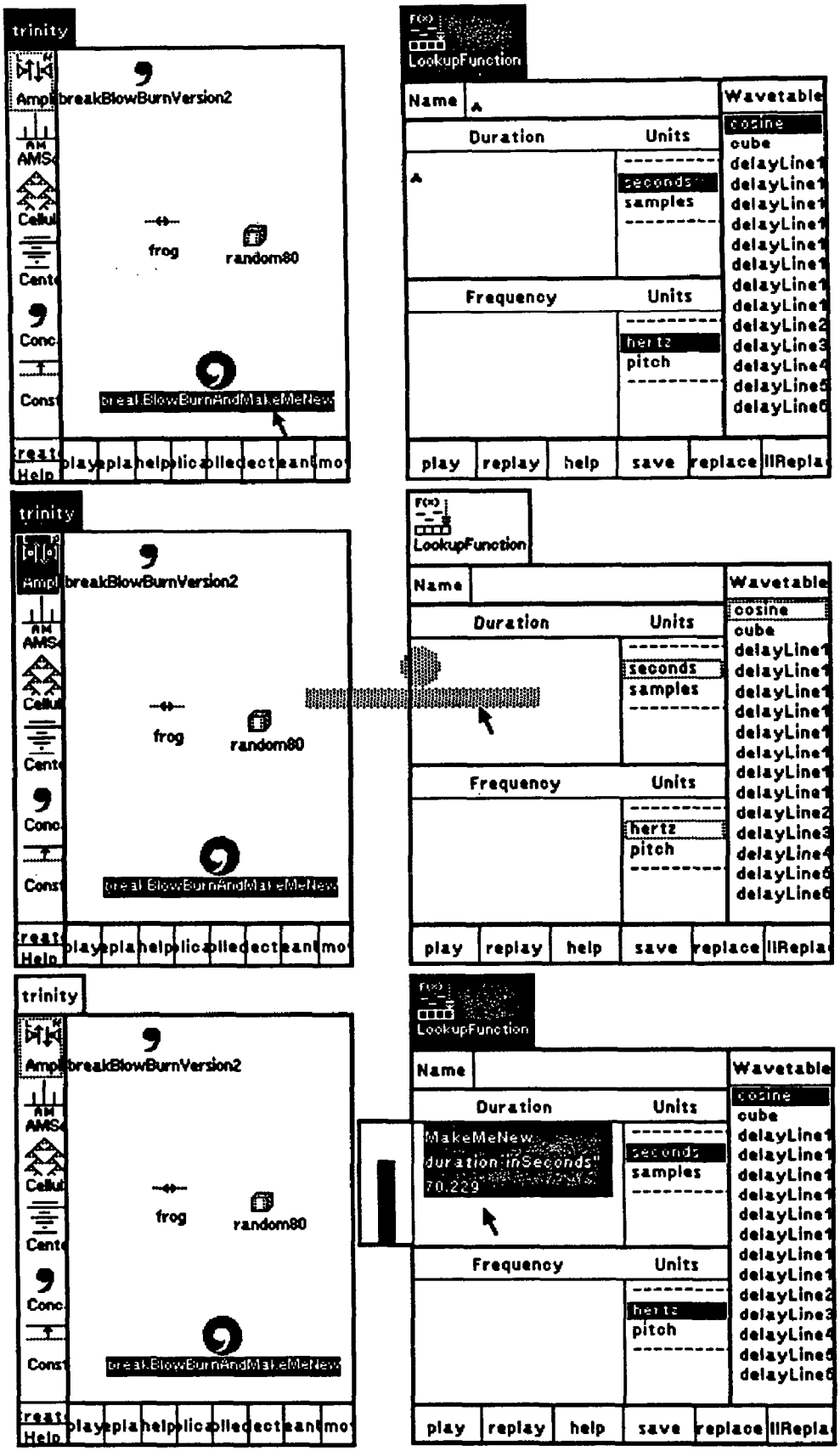

Figure 6. Dragging Between Views. I. A SoundPoint is selected, II. It is dragged across to the creation view, III. It is dropped into the LookupFunction's duration pane where it "spills its guts", yielding up its duration.

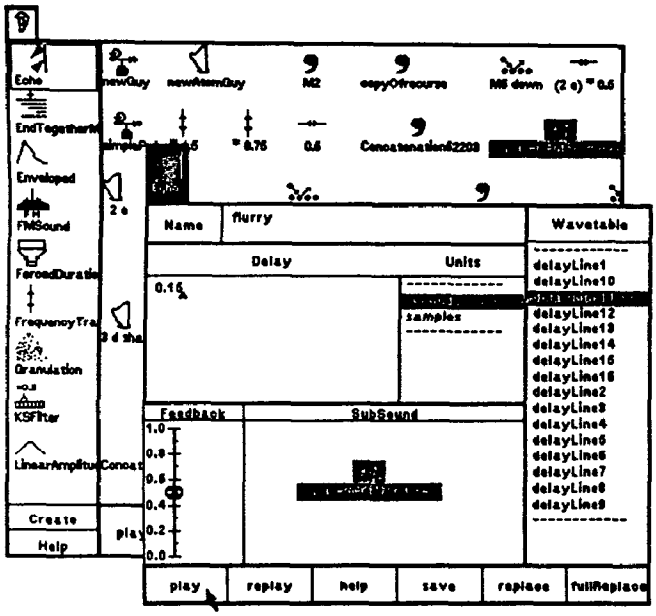

Figure 7. Echo Creation View. Parameters can be supplied from slider pots, selection from a list, typing, or dragging another SoundPoint into the pane.

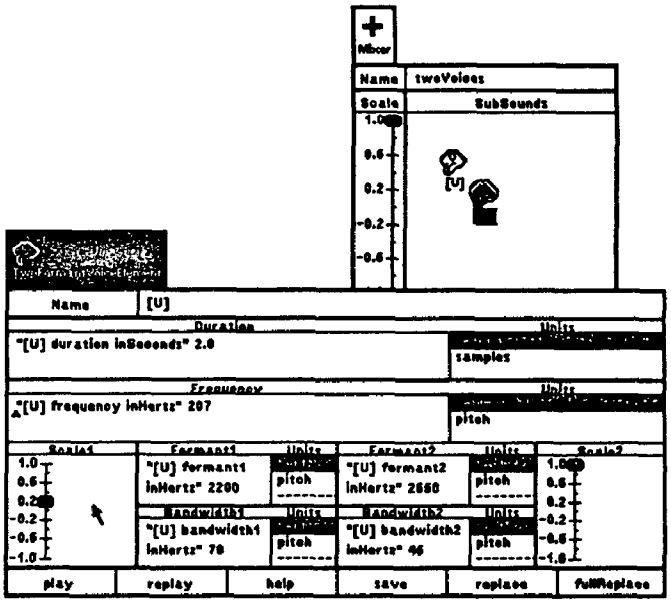

Figure 8. Opening up a Mixer reveals its subSounds. Any parameter of the subSounds can be altered and the new subSound can replace the old.

SoundPoint. For example, consider a SoundPoint representing a Mixer with two TwoFormantVoiceElements as its subSounds. Double-clicking that SoundPoint would cause a creation window to appear with the Mixer's parameters already filled in. Double-clicking on either of the *subSound's* SoundPoints would open a TwoFormantVoiceElement creation view with the appropriate parameters filled in as shown in Figure 8.

SoundPoints are opened in order to examine a Sound structure, to change some aspect of a Sound, or to clone a new Sound from an existing one.

1.4.4 Code Views

An alternative creation view, obtained by pressing the create button with the shift key down, is a CodeView containing a template instance creation method for the selected Sound class. Parameters can be filled in by hand, and SoundPoints in the boxes below can be referred to in the Smalltalk code as s1–s4 (Figure 9).

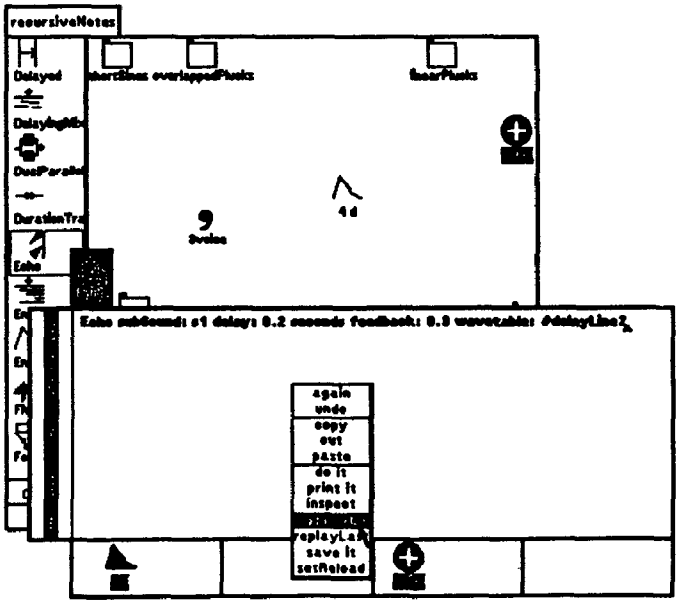

Figure 9. A CodeSound Creation View. The template parameters are replaced with actual parameters after which the code can be selected and evaluated. SoundPoints placed in the boxes below can be referred to in the code as s1-s4.

2. The Implementation of Kyma

The current implementation of Kyma was written using ParcPlace Systems Smalltalk-80 running on a Macintosh II.

2.1 The Representation of Sounds in Kyma

A Sound in Kyma is represented as a directed acyclic graph (DAG); a subSound node can be shared among several superSounds. A Sound DAG is similar to an expression tree in that the evaluation of the higher nodes depends on results of evaluating the lower nodes.

Figure 10 is an example showing an N-aryTransform — a Mixer with three subSounds: a FrequencyTransform and two Delays. Each of the Delays has a FrequencyTransform as its subSound, and so on. There is only one SoundAtom in this example, a LookupFunction. This LookupFunction is shared by three different superSounds.

From this example it's easy to see how traditional note-oriented music can be described in Kyma. The single Sound at the terminal node corresponds to the "instrument definition" of a synthesizer. FrequencyTransforms and DurationTransforms are used to obtain the different pitch and duration of each note and the delay times supplied to the Delayed nodes place the notes relative to one another either sequentially or as chords. At the CERL Music Project, this method is used to translate pieces composed on the IMS synthesizer into Sounds so that they could be played by Kyma on the Platypus. (The Interactive Music System, IMS, is a digital-synthesizer-based system developed at CERL prior to the development of the Platypus [Haken84], [Scaletti84].)

2.2 Samples Generation

2.2.1 Generating Samples in Smalltalk-80

Recall that a Sound can be described as a stream of samples; thus one could implement samples generation in Smalltalk by defining each synthesis algorithm as a Stream which responds to the message, **nextSample**. In general, a superSound would ask each of its subSounds for the subSound's next sample before returning its own next sample. In the case of a Mixer, for example, the **nextSample** method would be something like:

```
^ subSounds inject: 0 into: [ :sample :sub | sample + sub nextSample].
```

This was, in fact, the way samples generation was done in the first version of Kyma. However, Kyma was not capable of generating samples at a continuous rate of 20,000 samples per second. In fact, for even moderately interesting Sounds, it could take a minute or longer to compute each second of sound (i.e. 3 milliseconds or longer per sample).

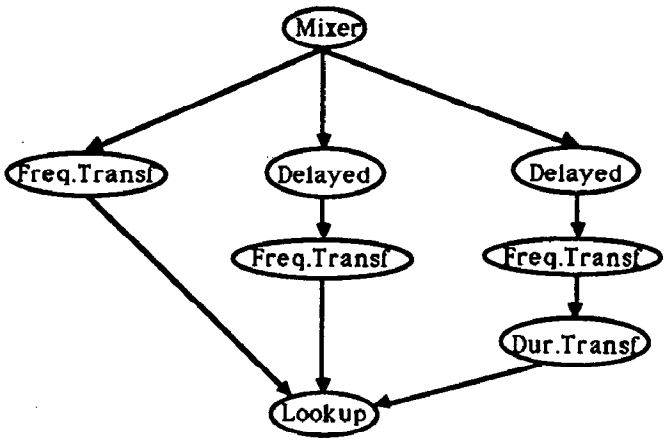

Figure 10. A Simple N-ary Transform Structure in Kyma.

2.2.2 Generating Samples on the Platypus

For this project, the speed problem was solved by moving that one crucial method, the **nextSample** method, onto a specialized piece of hardware — a microprogrammable digital signal processor called the Platypus [Haken87]. The Platypus is a *signal processor*, not a MIDI synthesizer; a signal processor has the distinct advantage of being programmable, thus allowing for the addition of new synthesis algorithms and the improvement of existing algorithms.

Designed by Lippold Haken and Kurt Hebel of the CERL Music Project, the Platypus can execute 20 million instructions per second (20 MIPS). To put that in some perspective, consider that the 68020 processor in the Macintosh II has a 3 MIPS peak and averages around 1.5 MIPS. For this reason the responsibilities have been factored between the two machines, giving each the job it can do the best. The abstract structure of the Sounds is handled by Smalltalk on the Macintosh, and the low-level samples generation is handled by the Platypus.

2.2.3 Assembly Language and Platypus Microcode

How does the speed of the **nextSample** methods written in Platypus microcode compare with the speed of those same methods implemented as 68020 assembly language primitives? For comparison, we used the Mixer algorithm given in 2.2.1. Assuming a clock speed of 16 megahertz, the assembly language implementation requires, in the best case,

```
1.125 + (3.9375 * number of subSounds) microseconds per sample
```

and in the worst case,

```
2.25 + ( 5.8125 * number of subSounds) microseconds per sample.
```

The implementation in Platypus microcode requires

.55 + (.5 * number of subSounds) microcoseconds per sample,

nearly an order of magnitude faster than the 68020 assembly language implementation.

Consider a concrete example: At a sample rate of 20000 samples per second, a maximum of 50 microseconds can be used to compute each sample. On the 68020, a Mixer with 10 subSounds implemented in assembly language would require, on average, about 50 microseconds per sample — just within the time constraint; however this leaves no time to compute the subSounds' samples, so a Mixer of 10 subSounds could *not* be computed in real time. On the Platypus, the Mixer would require less than 6 microseconds per sample, leaving more than 44 microseconds left over for the computation of 10 subSound samples. Since a LookupFunction requires fewer than 1.5 microseconds computation time per sample, a sample for a Mixer of 10 LookupFunctions could be computed in real time with about 30 microseconds to spare. In fact, Mixers of up to 32 LookupFunctions have been played using the Kyma/ Platypus system with its rather conservatively written microcode; optimization of the microcode would allow us to squeeze out even more simultaneous Sounds.

This speed difference convinced us that the Platypus was the best means for providing interactive playback for compositions of nontrivial complexity. Nevertheless, with the addition of a hard disk for storing samples, a version of Kyma with assembly language primitives could be used to generate samples for *delayed* playback (with some simple Sounds computable in real time), and this alternate solution may be implemented at some time in the future.

2.3 Sound objects on the Platypus

The Platypus solves the speed problem while introducing yet a different problem — how to get from the abstract DAG representation of a Sound in Kyma to a microcode representation of that same sound on the Platypus. The solution to this problem is to use an object-oriented microcode program on the Platypus.

2.3.1 The Platypus Hardware

The Platypus consists of:

- Three 16-by-16 multiplier-accumulators
- 1 megaword (16-bit words) of slow memory (250 ns access times)
- 1024 32-bit registers (50 ns access times)
- 2048 80-bit words for the microcode program

Each Platypus instruction takes 50 nanoseconds; thus at a sample rate of 20 kHz, up to 1000 instructions can be used to compute each sample. Input and output of sound is accomplished using stereo 16-bit analog-to-digital and digital-to-analog converters.

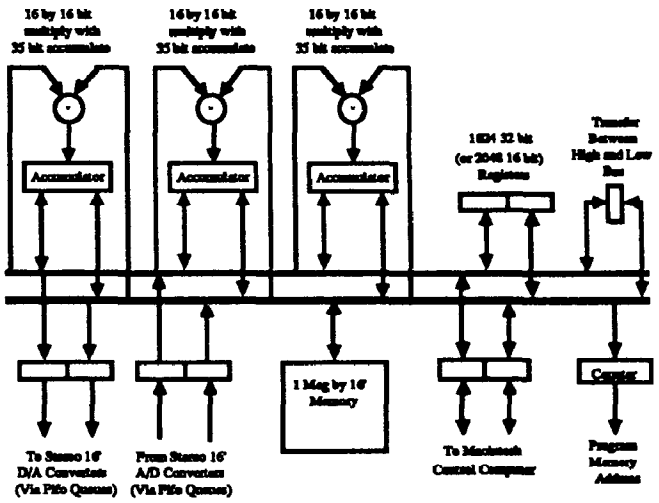

Figure 11. The Platypus Signal Processor [Haken87].

2.3.2 The Structure of Sounds and MicroSounds

Just as a Sound object in Smalltalk-80 is a structure containing instance variables and a pointer to its class and methods, that Sound's representation on the Platypus (called a microSound) is a collection of values in *registers* with a pointer to its *microcode* class and method. Since there is, in fact, only one method, the **nextSample** method, associated with each class, a microSound is simply a collection of registers with a pointer to its **nextSample** method in the microcode. In a sense, then, the Platypus microcode is an object-oriented program in which there is only one message ever sent (an implicit **nextSample** message) and only one value ever returned (the next sample in that microSound's stream).

This object-oriented program could be implemented by storing the Sound DAG in the registers of the Platypus and storing the the microcode methods and the control loop in the microcode memory. However, on any particular sample, many of the nodes of a Sound DAG are inactive; that is, each subSound has a finite duration, and not all of them are playing at once. Ideally then, the Platypus registers should contain not a *static* DAG, but a DAG that is growing and shrinking with time depending on how many of its Sounds are active on the current sample. This optimization speeds things up by decreasing the length of the microcode loop, and it it allows more complex microSounds to be stored in the Platypus' limited register memory.

In order to create a dynamic version of the DAG, it's necessary to know when Sounds turn on, when they turn off, and when there is a change in the number of a Sound's active

subSounds. This information is contained in an *event list*, a time-tagged list of changes to be made to the dynamic microSound DAG. Such a list is easy enough to obtain when each Sound knows its duration, its relative startTime, and all of its superSounds.

2.3.3 A Sound's Representation on the Platypus

Figure 12 illustrates the function of each part of the Platypus when it is running the Kyma microcode. The active microSounds are arranged in a circular linked list in the registers; changes to the microSound list are specified by the event list which is stored in memory. Each event consists of inserting a new microSound into the list or deleting a microSound whose "time is up".

A control loop in the microcode traverses the list of microSounds once per sample. For each microSound it does the following:

- Puts the address of the microSound's first register into the base address register.
- Follows this microSound's pointer to its **nextSample** method in the microcode.
- Computes the **nextSample** for this class of Sound using this microSound's values for any variables in the **nextSample** microcode and obtaining any subSound values from the sample stack.
- Pushes this microSound's next sample onto the sample stack.
- Changes the base address to the context of the next microSound in the list.

On the completion of each traversal of the microSound list, the microcode does the following:

- Outputs the sample at the top of the stack.
- Receives any data being written from Smalltalk
- Executes any events scheduled for this sample.

2.3.4 The Translation from Sound to MicroSound

When a Kyma Sound receives the **play** message, it extracts a chronological event list from the DAG in the form of time-tagged Platypus register operations. This event list is then downloaded to the Platypus memory. Once the event list has been downloaded, the Platypus is given a start signal; the Platypus then loops through the microcode, generating a sample each time through the loop and stopping itself when the event list has been exhausted.

2.3.5 Assembling the Platypus Microcode Program

Each Sound class which is not defined purely in terms of existing Sound classes implements an **assembleUsing:** method. This method contains the register definitions and the microcode assembly language implementation of the Sound's nextSample algorithm. Each time the machine is turned on or a change has been made to the microcode, a new microcode program is assembled by piecing together the fixed portions of the microcode (e.g. initializations, the control loop, some register definitions, etc.) with the

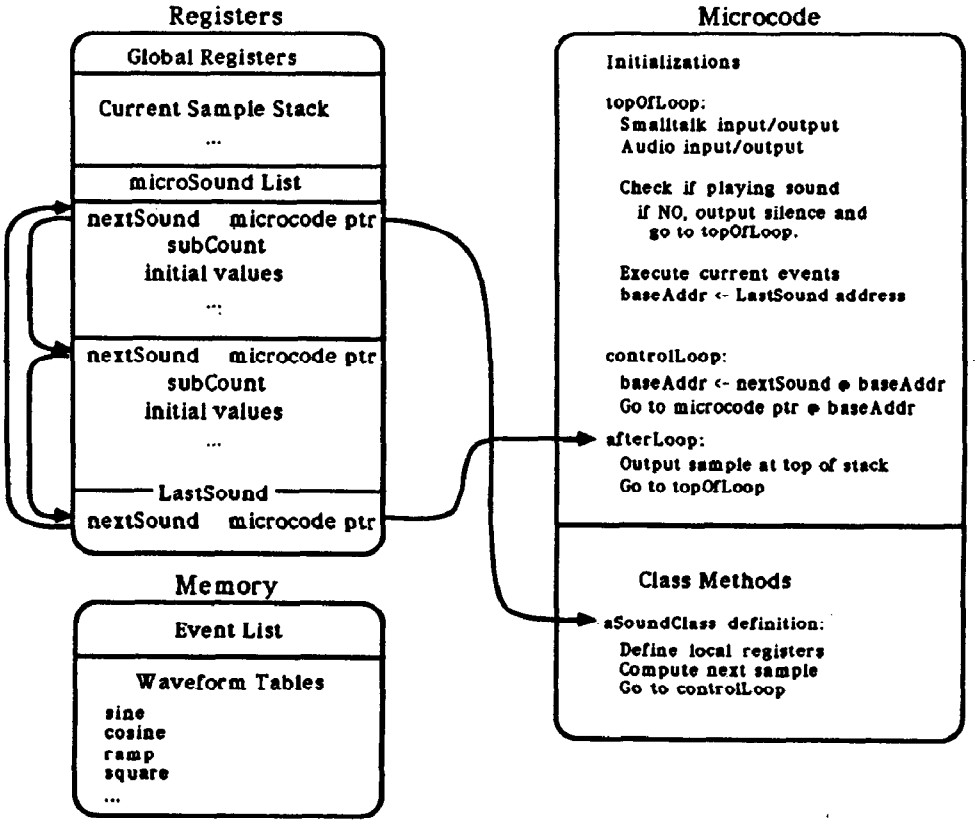

Figure 12. A Block Diagram of the Platypus Running the Kyma Microcode.

portions of microcode supplied by each Sound class in its **assembleUsing:** method. This assembled microcode is then downloaded to the Platypus where it runs until the machine is turned off (unless, of course, the user decides to change the microcode program and reassemble it).

2.4 Microcode vs MicroSounds

In contrast to the microcode, which is downloaded only once, a new microSound is downloaded each time a Kyma Sound receives the play message. The time between telling a Sound to play and actually *hearing* that Sound is spent in Smalltalk generating the event list; once the event list has been downloaded to the Platypus, the samples generation takes place in real time. (For this reason, *replaying* a Sound is virtually instantaneous; the Platypus is simply sent the start signal and it replays whatever event list it has currently in memory).

The size of the microcode is relatively fixed, changing only when a Sound class is added or modified; the size of the microSound DAG in the registers is related to the complexity of the Sound as it was defined in Kyma. In this context, complexity is related to the number of on/off events and not to the duration of the Sound in real time, i.e. a sine wave lasting for 2 hours is less complex than a 2 second flurry of several sine waves turning on and off at different times.

In some sense, the microcode **nextSample** methods on the Playpus can be thought of as the instructions of a language, and the particular sequence of these instructions (inherent in the structure of each Sound object) as a program. Thus in Kyma, a music composition is a program.

3. Conclusions

3.1 Smalltalk as a Music Language

One of the goals of Kyma is to provide a uniform and flexible structure which does not impose stylistic assumptions upon the composer. The uniformity of objects in Smalltalk has made it an ideal environment for the development of such a flexible structure.

3.2 Other Object-oriented Music Languages

Smalltalk seems to have a disproportionately large number of adherents in the computer music community. For details on other music-related work in Smalltalk, the reader is referred to the articles by Krasner, Pope, and Lentczner listed in the references. The paper by Cointe et al describes FORMES, a music language in use at IRCAM which was developed in VLisp, the paper by Polansky et al is a discussion of the HMSL language written in an object-oriented FORTH environment, and the paper by Gary Greenberg outlines his Object LOGO music environment.

3.3 Future Plans

Kyma was intended for use by composers of experimental music, but it is general enough to have other applications as well; it could, for instance, be used as part of a audio signal-processing workstation or in the design of psycho-acoustic tests.

The next large step in the development of Kyma depends on another Smalltalk-80 application which is currently under development. The Javelina environment [Hebel87] can generate microcode for any of several signal processors from a mathematical function specification. At some point, each of Kyma's **nextSample** algorithms will be replaced by an equivalent mathematical function. Javelina will be used to generate the microcode for whatever signal processor is being used. In fact, given a model of the 68020, Javelina could also be used to generate assembly language primitives in those cases where no signal processor is available.

It is hoped that Kyma will never actually be "completed" but will serve as a continuously evolving set of tools for the author's experiments in music and structure.

4. Acknowledgments

This work received generous support in the form of a fellowship from Apple Computer and the InterUniversity Consortium on Educational Computing. The Computer-based Education Research Laboratory at the University of Illinois should be acknowledged for its continuous support of computer music research at the CERL Music Project since 1974. Innumerable discussions with Kurt Hebel were both entertaining and highly influential in the development of Kyma.

5. References

Buxton, W., W. Reeves, R. Baecker, L. Mezei. 1985. "The Use of Hierarchy and Instance in a Data Structure for Computer Music." In C. Roads and J. Strawn, ed. *Foundations of Computer Music*, MIT Press: 443-466.

Cointe, P., J.P. Briot, B. Serpette. 1987. "The FORMES Language: a Musical Application of Object Oriented Concurrent Programming." In A. Yonezawa and M. Tokoro, ed. *Object-Oriented Concurrent Programming*, MIT Press: 221-258.

Goldberg, A. and D. Robson. 1983. *Smalltalk-80: the Language and Its Implementation.* Reading, Massachusetts: Addison-Wesley.

Grossman, G. 1987. "Instruments, Cybernetics, and Computer Music." In J. Beauchamp, ed. *Proceedings of the 1987 International Computer Music Conference*, Computer Music Association: 212-219.

Greenberg, G. 1987. "Procedural Composition." In J. Beauchamp, ed. *Proceedings of the 1987 International Computer Music Conference*, Computer Music Association: 25-32.

Haken, L. and K. Hebel. 1987 "The Platypus Programmers' Reference Manual" Technical Report. Urbana: University of Illinois Computer-based Education Research Laboratory.

Haken, L. 1984. "A Digital Music Synthesizer." M.S. thesis. Urbana: University of Illinois Department of Electrical Engineering.

Hebel, K. 1987. "Javelina: An Environment for the Development of Software for Digital Signal Processing." In J. Beauchamp, ed. *Proceedings of the 1987 International Computer Music Conference*, Computer Music Association: 104-107.

Hebel, K. and R.E. Johnson. 1988 "Arithmetic and Double-Dispatching in Smalltalk-80", in preparation.

Krasner, G. 1980. "Machine Tongues VIII: The Design of a Smalltalk Music System." *Computer Music Journal* (4) 4: 4-14.

Lentczner, M. 1985. "Sound Kit: a Sound Manipulator." In B. Truax, ed. *Proceedings of the 1985 International Computer Music Conference*, Computer Music Association: 237-242.

Loy, G., C. Abbott. 1985. "Programming Languages for Computer Music Synthesis, Performance, and Composition." *Computing Surveys* (17) 2: 235-265.

Polanksy, L., D. Rosenboom, P. Burk. 1987. "Overview (Version 3.1) and Notes on Intelligent Instrument Design." In J. Beauchamp, ed. *Proceedings of the 1987 International Computer Music Conference*, Computer Music Association: 220-227.

Polanksy, L., D. Rosenboom. 1985. "HMSL (Hierarchical Music Specification Language) A Real-Time Environment for Formal, Perceptual and Compositional Experimentation." In B. Truax, ed. *Proceedings of the 1985 International Computer Music Conference*, Computer Music Association: 243-250.

Pope, S. 1986. "The Development of an Intelligent Composer's Assistant: Interactive Graphics Tools and Knowledge Representation for Music." In P. Berg, ed. Proceedings of the 1986 *International Computer Music Conference*, Computer Music Association: 131-144.

Pope, S. 1987. "A Smalltalk-80-based Music Toolkit." In J. Beauchamp, ed. *Proceedings of the 1987 International Computer Music Conference*, Computer Music Association: 166-173.

Pope, S. 1988. "The HyperScore ToolKit and Software Tools for Computer Music." *Hoopla!* (1) 2: 27-29.

Pope, S. 1988. "Building Smalltalk-80-based Computer Music Tools." *Journal of Object Oriented Programming* (1) 1: 6-11.

Rodet, X. and P. Cointe. 1984. "FORMES: Composition and Scheduling of Processes." *Computer Music Journal*(8) 3: 32-48.

Scaletti, C. 1987 "Kyma: an Object-oriented Language for Music Composition." In J. Beauchamp, ed. *Proceedings of the 1987 International Computer Music Conference*, Computer Music Association: 49-56.

Scaletti, C. 1984 "The CERL Music Project at the University of Illinois." Computer Music Journal(9) 1: 45-58.