Kyma: An Object-oriented Language for Music Composition

Carla Scaletti
CERL Music Project
252 Engineering Research Laboratory
103 South Mathews Street
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801-2977 U.S.A.
https://uic.edu/uic/earl/scacellti

Abstract
Kyma provides the composer with a mechanism for manipulating the unwieldy large amounts of data associated with digital audio synthesis without resorting to models based on written scores and traditional instruments. In this language, everything, from the structure of a single sound to the structure of an entire composition, is viewed as an object to be composed.

Kyma is written as a set of extensions to Smalltalk-80 on the Macintosh. Smalltalk methods are used to produce sound in one of two ways: by generating individual samples and sending them to the Macintosh's 8-bit digital-to-analog converter or by generating a microcode which is used to compute the samples on the Platypus, a digital audio signal processor developed by the CERL Music Group.

1. Motivations
As is the case with many composers, I was initially drawn to the computer in order to continue and expand upon what I had already been doing for several years by hand: using algorithms and models in music composition. At the University of Illinois I was initiated into the mysteries of software synthesis à la MUSIC48/560 and came to know the pleasures of analog-circuit-model timbre design and PORTTRAN-generated event lists. Although I became addicted to the thrill of designing the sound in such a direct way, I developed no such attachment to the week-to-month long turnaround times. During this period of time, I visited the audio lab of a friend of mine who was doing AI research down the hall from me, the Xerox Dolphin, and I got it into my head that someday I would have such a machine on which to do real music composition.

Working with a digital synthesizer provided an opportunity to concentrate on the algorithmic aspects of composition but became increasingly dissatisfying for two reasons: one is that I kept running into a wall between structure and timbre -- between "score" and "instruments". It became clear that composition and synthesis had never really been separate issues; the other reason was that I found myself spending too much time circumventing the rotational bias of the input language.

I imagined a language in which timbre and event lists were part of a continuum of structure, where the composer could choose to think in terms of notes and keyboards and staves but in which this structuring would be no easier and no harder to implement than any of countless, as yet unevolved, alternatives. I saw the microprogrammable signal processor and disk system being developed by Lippold Haken and Kurt Hebel as a way to regain the flexibility of software synthesis while retaining synthesizer-like response times. When Smalltalk-80 became available for the Macintosh, it seemed that I could at last have my Dolphin-like environment. I have greedily tried to put all these pieces together into a system which I call Kyma (because it is, in a fanciful sense, a tool for exploring wave phenomena).

2. Introduction
For several years now, there has been an interest in creating music languages using Smalltalk [Kramer80], and, more recently, Smalltalk systems have been developed for the editing of digitally recorded sounds [Lenzer85] and the manipulation of event lists [Pope86]. Other object-oriented music languages include FORMES, writes in an object-oriented LISP [Rodet84], a development system called MUSACT [Morris85], and MAX which utilizes the 4X digital signal processor [Pincus86]. Another language which is related to Kyma, in that it has as its basis a hierarchical model for musical structure, is IIMSL [Polanek85].
Kyna was created to the following specifications:

- A mechanism was desired whereby the vast amounts of data necessary in digital synthesis could be packaged into conceptual chunks (objects) by the composer, i.e. the language itself would not impose relational or stylistic preconceptions.

- Feedback times were to be as short as possible, since it is a language for composition and not performance, real-time response was not required.

- The structure of a composition was to be, itself, an object, a record of all the transformations applied to its constituent atoms. In other words, transformations of a sound were not to be destructive modifications of the sound but to function as filters (with which to view the sound) in a new way.

- It was desirable that work done in this environment be cumulative and that the details of the system could be learned incrementally.

- It was desired that the composition of sounds could be done graphically and that each sound object could be viewed in multiple representations.

- Also desired was the means by which the composer could create a "sound universe", endow the sound objects in this universe with certain properties and relationships, and explore this universe in a logically consistent way.

The first step in this project was the invention and refinement of the class hierarchy and the development of Smalltalk methods for doing samples generation on the Macintosh. Next, it was necessary to come up with a way in which to represent a Kyna structure on the Macintosh as microcode on the Platypus so that the Platypus could take over sample generation. Future plans include the development of a graphic interface and the addition of logic to the Sounds.

![Graphical representation of sound objects and relationships](image)

Figure 1. Partial Sound Class Hierarchy for Kyna. The class hierarchy shows the inheritance of properties and relations.

![Diagram illustrating sound transformation and substitution](image)

Figure 2. An Example of Substitution. Sound A is substituted for Sound B resulting in Sound C.

3. Class and Instance Hierarchies

In Kyna, all objects are instances of class Sound, of which there are two subclasses, SoundAtom and SoundTransform (Figure 1). Sound can be recursively defined as:

1. *S*(t) = a function `f` of *S*(t-1)

A SoundAtom is just like atoms used to be, unchangeable and indivisible. A SoundTransform is an alternative view of one or more subSOUNDS; it does not change its subSOUNDS. It is an object in and of itself, something like a prism, providing a transformed view of whatever is observed through it.

3.1 Instance Hierarchy

This class hierarchy is relatively fixed, i.e. it can only be changed by someone who knows Smalltalk. A composer working with Kyna will spend much more time designing instance hierarchies (of which the Sounds in Figure 2 are examples) which are defined by the relations, subSound and superSound.

3.2 Substitution

Every instance of Sound, from a timbre to an entire
composition, is a Sound object in Kyma and can be substituted for any other Sound object. For an example, see Figure 2.) Substitution is interesting in that a structural family of compositions could be composed by substituting different SoundAtoms into the same SoundTransform.

An appealing feature of the Kyma environment is that composers may begin producing complex sound structures almost immediately simply by taking previously constructed structures and applying substitution to them. (The same has been noted of another such integrated environment, FORMOSA (Roden4)) A composer can learn the details of Kyma as they are needed; the next step would be to learn how to define new Sounds. Of course, in order to take full advantage of the system, the composer would go on to learn Smalltalk and the Flaypups assembly language; however, this too may be undertaken incrementally, starting out by altering copies of existing code. Note that this is in no way an endorsement of the "no one should have to learn anything to do computer music" philosophy. As with any language, the use of Kyma in experimental composition will require careful thinking and a substantial investment of time; it is hoped, however, that Kyma will assist the composer's exploration and not create, as some "easy" languages do, insurmountable roadblocks to experimentation.

3.3 Timelessness

In Kyma, every sound believes that it starts on sample 1 and continues for its duration; in order to obtain sounds with staggered start times, a UnaryTransform called DelayedSound is used. A DelayedSound's subSound still thinks it starts on sample 1, but its enveloping superSound doesn't tell it that sample 1 has occurred until it has waited for a specified delay time. A Sound, while it is in Kyma, may contain cycles and shared nodes, but before it can be played, it must make a copy of itself as a tree. It is only at this point, when it is about to be played, that a Sound becomes fixed in time, and each of its subSounds is assigned an absolute starting sample.

4. Samples Generation

In Smalltalk-80, the central paradigm is the message-send, and the most important message in Smalltalk sample generation is newSample. Figure 3 contains the Smalltalk methods which would be invoked if an instance of Mixer whose subSounds were all instances of StaticCosine were asked to return its next sample.

Figure 3. Sample Generation in Smalltalk. Assume that a Mixer receives the message nextSample. Stacks Meta is in a subclass of Sound, it inherits Sound's method for considering, shown at the top of the figure. This method returns a value of 0 if the Mix is 0, or 1 if it is not; it needs until the message next. Note that each subclass override Sound's method for next. In Mixer, the next method needs the message nextSample to each of its subSound, returning the sum of the return. If the subSound were all of class StaticCosine, then StaticCosine's method for next would compute and return the next sample returning to a recurrence relation (Smith85).

SoundTransform: 'accompany'
nextSample

"If the receiver sound is 0, return 0, if the message to get and return the
next sample in the stream; if not, return a sample if 0. In either case, increase
the receiver's private time by one:"

| x <- privateTime | y <- 0 |
| if: [true | if: [false | x <- x + 1 |]

SoundTransform: 'private'
next

"Each subSound computes the next sample in the stream directly."

self subSoundResponsibility

MixerMethod: 'private'
next

"Add the results of asking each subSound for its nextSample, and return
the sum as the current sample. This message is only sent to Sounds which are
active at the time."

| x <- subSounds injDrop: 0 | y <- subSounds injDrop: 0 | "Delaysound" injDrop: 0 | "DelaySound" injDrop: 0 |

StaticCosine: "Subclass: 'private'
next

"This message is only sent if the receiver is active. Compute the next
value of the StaticCosine using a recurrence relation described by Julius Smith
in the Proceedings of the 1983 ICIC."

| x <- sampleAmplitude | y <- privateTime | z <- (privateTime | 0) |
| x <- if: | x <- if: |
| | x <- if: |

1987 ICIC Proceedings 51
5. Microcode and MicroSounds

As might be imagined, samples generation in Smalltalk is unbearably slow. For example, a StaticCosine lasting 3 seconds at a sample rate of 20 kHz requires 3 minutes to compute. In that same 3 minutes the Platypus digital signal processor can execute 3.6 billion instructions. Clearly, it makes more sense to let the Platypus generate samples and to use the Macintosh and Smalltalk-80 for dealing with the Sounds on an abstract level; thus the next phase in this project was to come up with a way in which a structure in Kyma could generate a microcode representation of itself for the Platypus, a representation which would retain some of the object-oriented nature of the original.

5.1 Platypus Hardware

Designed specifically for digital audio signal processing, the Platypus is part of a system which includes a Macintosh controlling computer, an 80-megabyte hard disk for sound samples (Henry's), stereo 16-bit ADC's and DAC's. It consists of:

- three 16-by-16 multiplier-accumulators
- one megabyte of 16-bit words intended as a waveform and function memory
- 1024 32-bit words of register memory
- a microcode program memory which can store 2048 80-bit microcode instructions

Each instruction takes 50 nanoseconds, thus at a sample rate of 20 kHz, an algorithm may use 1000 instructions to compute each sample. Figure 4 shows a block diagram of the Platypus.

5.2 Microcode

Initially, a method was added to each Sound definition which would return a new Sample algorithm in a pseudo-microcode assembly language. In order to play a SoundAtom, this pseudo-code was assembled into microcode and embedded in a standard microcode loop. This entire construction was then downloaded to the Platypus microcode memory where it would generate samples in an infinite loop. (Once the Sound's duration was exceeded, it would continue outputting sample values of 0 forever.) Figure 5 provides more details on how this worked.

In order to play a SoundTransform, a microcode was constructed recursively with code for each subSound embedded in that of its superSound. (See Figure 6.)

Figure 5. Constructing a Playable Microcode for a SoundAtom—The code for the SoundAtom is embedded in a loop.

Figure 6. Representation of a SoundTransform in Platypus Microcode. Note that each variable name is tagged with a number unique to that Sound allowing for multiple instances of a single class.

52 1987 ICMC Proceedings
In this first version, each instance of a class contributes all of the code necessary to compute its next sample; if there are several instances of a single class, each instance contributes code which differs from that of the others only in the names of the variables. In addition, the code for each and every subSound must be present in the microphone from the start, no matter how long the delay before a particular Sound is to contribute anything but zero to the output sample. While this method produces microcodes which are relatively easy to understand and which produce the Sound exactly as intended, complex Sounds quickly exceed the microphone memory size. Some other means had to be devised, some way of reducing the degree of redundancy and of exploiting the fact that not all subSounds are active on every sample.

5.3 MicroSounds
The second solution required that the function of the three Platypus memories be restructured. Originally, the microphone memory was to contain only sample generation algorithms which were to be looped-through indefinitely. The waveform memory was to be used to store functions for table-lookup, and the register memory was to store values accessed by the microcode algorithms.

But there is another way in which these memories could be utilized. What if the microphone memory were to contain all of the Sound class definitions? Then each instance of a class could just have a pointer to its class' microcode rather than its own (redundant) copy of the code. In fact, all that is really needed to represent an instance of a Sound in this scheme is a set of initial values and a pointer to the Sound's class microcode; these sets of values can be stored in the register memory. But how can a single microcode method refer to variables which differ from instance to instance? This required the addition of a base address register to the Platypus hardware. Rather than referring to absolute registers, the microcode can now reference the variables that differ from instance to instance as offsets from the address stored in the base address register. Since MicroSounds (the Sounds) are now represented as chunks of data in the register memory, all that is required to load a new MicroSound is a sequence of register-write instructions. By commandoeering a portion of the waveform memory and subverting it for use in the storage of a sequence of time-tagged changes to register memory, it is possible to exploit the fact that not all MicroSounds are active on every sample. (Figure 7 provides more details on the sequence of control in this new scheme.)

Figure 7. The Function of the Register Memory (RM), the Microcode Memory (MM) and the Waveform Memory (WM) in Kyma. A control loop and the class definitions reside in MM; this program need only be changed when a modification is made to the class hierarchy in Smalltalk. Instances, in the form of initial values for the variables in the class definitions, are dynamically shifted in and out of RM. At the top of RM, a cursor/sample stack is maintained so that a superSound can have access to the current sample returned by each of its subSounds. While part of the one megabyte WM is still used for table lookup, a portion of it is reserved for the SubSounds, a list of time-tagged changes to be made to registers in RM. Since not every subSound is necessarily active for the entire duration of the top level Sound, subSounds which are not yet active on the current sample can be stored in WM waiting to be read in on a later sample. Notice that each MicroSound has two pointers stored in its first register. This pointer in the high

1987 ICMC Proceedings 53
This representation of Kyma structures on the Platypus is more object-oriented than the previous scheme. In Kyma, a Sound (like all Smalltalk objects) is a set of data with a pointer back to its class methods. On the Platypus, a MicroSound is a chunk of data in the register memory with a pointer to its single class method in the microcode memory. One could go as far as to say that there is still a degenerate sort of message-passing going on: the only message is an implicit *nextSample* and the only value returned by a MicroSound is its next sample which it deposits on the top of the current sample stack.

5.3.1 Playing a Sound on the Platypus

Figure 8 presents a detailed description of how a Kyma Sound is played on the Platypus. The Sound to be played is translated into a set of time-tagged changes to the register memory, a Schedule. If the class methods have already been downloaded to the microcode memory, the Schedule is downloaded to the memory and the Platypus is ready to play the Sound.

6. Future Directions

Figure 9, a screen print of the current version of a Kyma view, shows only the most rudimentary beginnings of what ought to be a primarily graphics-oriented system. It is planned, in the next phase of Kyma's development, to construct an interface of the sort described in [Desain86]; composing Sounds in Kyma ought to be done using a visual/convivial interface not, as it is now, using typed-in Smalltalk statements.

In the current version, any changes to the Kyma class hierarchy also require rewriting of the Platypus microcode. It would be ideal if, not only were the microcode generation automatic, but the microcode for any of several digital signal processors could be thus automatically generated, lending Kyma more machine independence. Javelina, another software environment under development at the CERR Music Project, addresses the problem of automatic generation of optimized microcode from a mathematical specification [Fender87]. Since Javelina, like Kyma, is being realized in Smalltalk, it is hoped that the microcode generating capabilities of Javelina might at some point be incorporated into Kyma.

Kyma has been set up to allow for a sort of composition which might be called the Society-of-Sounds or Extended-Game-of-Life; that is, rather than specifying the precise sequential combination of Sounds, the composer would instead specify the precise properties and relations between Sounds and microcode for all秦属Sound classes. The first action in method *makeMicrocode* is to ask Kyma whether the microcode has already been downloaded. Since the class definitions and control loop are relatively constant, a single instance of class Microcode is served, and this instance need be downloaded only once at the beginning of a listening session.

Next, Kyma is told to wait the Platypus so that this Sound's Schedule can be downloaded into WM. Once downloading is complete, Kyma is set to start up the Platypus.

A new microcode need only be assembled when a change has been made to the Sound class hierarchy. To initialize a new microcode, class Microcode creates a new instance of MicrocodeAssembler. After sending it the standard initialization of control microcode, each subclass of Sound is asked to send its class description. The instance of MicrocodeAssembler returns as array of microcode words as well as a dictionary listing the starting address of each class definition.

In order to return its Schedule, a Sound makes a copy of itself and removes all instances of DelayedSound by pushing the delay times cumulatively down to the leaves while assigning each Sound an absolute start time. This new object is called a *PPSSound*. A PPSound responds to the message, *eventList*, by returning a SoundCollection of time-tagged events classified as either output events or incremental movements of the Sound's *microCode*. These Event objects can then be translated almost directly into a list of changes, and this list of changes, the Schedule, is downloaded into WM on the Platypus.

Figure 9. A Screen Print of a Kyma View. Sounds coded in the lower right window can be sized in the Sound List (the narrow window on the left). When a Sound is selected in the Sound List, it can be played on the MainSound, or on the Platypus, displayed as an amplitude versus time plot in the upper right window, or have its *microCode* names at any level displayed in the upper right window.
then turn them loose to work it out among themselves. This is not an abdication of responsibility on the part of the composer but more a distribution of deterministic and logical behavior among all the objects in the universe of a composition.

6. Conclusion

Kyma is a language for experimental composition which, in conjunction with the Platypus digital signal processor, provides the composer with an interactive system for software synthesis without a notational or stylistic bias. Kyma provides an instance hierarchy mechanism allowing the composer to group any set of sounds into a conceptual chunk and, thereafter, manipulate it as a single object. The structure of a Sound is an object distinguishable from its constituent subSounds allowing for substitution of any Sound object for any other. Sound is produced either through sample generation and conversion on the Macintosh (extremely slow) or by the generation of microcode and MicroSounds for the Platypus Digital Signal processor so that it can generate samples. Future plans include the development of a graphical/conversational interface, the automatic generation of optimized macrocode, and the addition of logic to individual sounds. This paper serves as a progress report on a project not quite a year old. It is hoped that the structure of Kyma and the Smalltalk environment will prove flexible enough to withstand many years of compositional experimentation and further development.

7. Acknowledgements

I would like to thank Professor Ralph Johnson for his interest in this project and for the inside tips on how to speed up my Smalltalk code. To Lippold Haken, the designer of the Platypus, thanks for inviting a composer to be a part of the previously all-engineer CERL Music Project even though you knew that composers can be such troublemakers (and thanks for wiring in the base address register!) Kurt Hebel introduced me to Smalltalk in the first place; his insight and feedback in countless discussions have been entertaining, influential, and deeply appreciated.

8. References


