Carla Scaletti
Symbolic Sound Corporation
P.O. Box 2549
Champaign, Illinois 61825–2549 USA
scaletti@symbolicsound.com
www.symbolicsound.com

Computer Music Languages, Kyma, and the Future

Eric Lyon organized this symposium around the following two questions. Why has some computer music software survived and developed a following? Where is computer music software today, and where might it be headed in the future? Acknowledging that the term ''computer music'' is by now redundant, since virtually all music involves the use of computers in some form or another, he posed these questions with respect to computer music software that supports experimental music. Experimental music is not limited to any specific musical style or genre. An experimental musician is one who approaches each act of musical creation in a spirit of exploration and innovation, often with the goal of inventing new kinds of music that have never been heard before.

I would like to take Eric Lyon's refinement a step further by observing that all of the software examples included in this symposium (along with some others that are not represented here) belong to a special category of computer music software called computer music languages. Most software packages can be classified as utilities; they perform a well-defined, familiar function that is needed by a large number of people. A software package that emulates all the functions of the traditional multi-track recording studio would be one example of a utility. But the pieces of software that Eric Lyon has chosen to include in this symposium are different; they are examples of computer music languages.

A language provides one with a finite set of ''words'' and a ''grammar'' for combining these words into phrases, sentences, and paragraphs to express an infinite variety of ideas. A language does not do anything on its own; one uses a language to express one's own thoughts and ideas. This is what makes these particular software packages so open, extensible, and useable in ways unanticipated by their authors. And that is why, although they may never command the same market share as utilities, they have had a longer-lasting and deeper influence on the evolution of music.

This article is organized into three sections. The first section is an identification and discussion of factors that can contribute to the success and longevity of a computer music language. In the middle section, I try to illustrate some of those factors (like extensibility) using specific examples from the Kyma language. The last section is a speculation on the role computer music languages could play in a future world where art, the economy, and human beings are very different from the way they are today.

Factors Contributing to the Success of a Language

Why have some computer music languages survived over the years and attracted a sizeable number of users? Although we would like to believe that the only reasons for the success of these languages are their inherent attributes, many of the factors contributing to success have little to do with the language or technology itself. There are, for example, the lucky accidents of geography, economic/social class, and chromosomal makeup. Given these ''accidents'' of birth and fortune, what other necessary (but not sufficient) factors can contribute to the longevity and acceptance of a computer music language?

Reasons Intrinsic to the Language

Several factors intrinsic to a language contribute to its relative success. *A language is successful if people are using it successfully.* It does not matter how elegant or beautiful a language is in theory or on paper if no one is using it. Any language that has longevity and a following also has a corresponding list of creative projects that have been successfully completed using that language.

Computer Music Journal, 26:4, pp. 69–82, Winter 2002

A language is successful if it answers a need that is not otherwise satisfied. How does a language gain users in the first place? First it must adequately address some basic needs, and then it must also answer some need (tangible or otherwise) that is not currently being satisfied elsewhere.

A language is successful if it is able to express the unanticipated. In a computer music language, just as in a natural language, it should be possible to say something new. For example, a highly restrictive data-flow editor—while it may protect one from mistakes—also prevents one from discovering the sounds that its author could not imagine. A successful music language does not impose a particular musical style upon its users; it must be flexible enough and abstract enough to generate musical utterances in any style and in any genre.

A language is successful if its underlying data structure can support extensions and multiple interpretations without violating the original model. This ability to express the unanticipated extends beyond the users of the language to the developers of the language. The underlying data structures of a successful language should make it easily extensible, even in directions that the designer had not originally anticipated. When a language has this characteristic, it can evolve and adapt.

A language is successful if its author can strike a balance between providing users with what they say they need and what they do not yet know that they need. It comes as no surprise that when the author of a computer music language is willing to make adjustments and additions to the language in response to user feedback that those languages become more successful and develop a loyal following. But the feedback process is not quite as direct as it might seem at first glance. If an author simply went through the ''wish lists,'' item by item, adding each feature that was asked for, the result would be a bloated and jumbled mass of features without the logic or framework necessary for the user to make sense of it all.

Instead, the authors of the successful languages have had to digest and analyze the feedback first before making modifications to the language. They have had to combine all the specific requests, generalize them, and produce meta-solutions that satisfy several of the issues at once. Even more important is the task of giving people what they do not yet know that they need. People do not always know that they want something if they have never seen it before. But that is often exactly what they want and need most of all: something new.

A successful software designer must maintain a balance between responding to user feedback and maintaining a consistent vision for the language. At times, it is the users who are specifying the language and at others, it is the author who forges ahead into new territory. This is not so different from a composer who gives the audience some familiar material while also nudging them along into new musical territories.

Reasons Extrinsic to the Language

A language is successful if one can learn it and learn from it. Whereas the success of a utility depends in large part on its being completely obvious and easy to use without having to refer to the manual, computer music languages, by definition, require more of their users. Simply knowing the syntax of a language does not automatically give one something interesting to say. The most successful users of computer music languages are those who also take an interest in learning more about sound, music, and structure. For a computer music language to be successful, there must also be opportunities for learning more about sound and music creation in the context of that language: workshops, courses, tutorials, extensive documentation (via print and other media), books, lectures, an online forum, technical support, etc.

One of the beneficial side effects to learning a computer music language is that the basic principles are completely transferable to other languages and even to other utilities and hardware setups. Learning a computer music language provides one with a basic understanding of sound, logic, and structure that one can apply no matter how technology evolves in the future. By contrast, the knowledge of which buttons to press in a special-purpose utility becomes obsolete as soon as that utility is upgraded or replaced.

A language is successful when it has a community of users. Although an artist is almost by definition independent, there is also some stimulation and practical assistance to be derived from interacting with other people who use the same language. Successful languages have groups of users who provide technical support, new developments, education, and appreciation for each other.

A language is successful when it serves as a nexus for interdisciplinary cross-fertilization. The most powerful function of any language (whether it is a programming language, a computer music language, or a natural language) is to act as a nexus for the cross-pollination of ideas among different fields. Each person who uses a language also has some effect on its development. When a language is general enough to satisfy the needs of users having different backgrounds, different goals, and different knowledge sets, it can also act as the vector for transferring ideas and protocols from one discipline into another. When the ideas and protocols are applied in the new field, they acquire additions and alterations that in turn find their way back into the language, and so on.

A language is successful when its author uses it regularly. Using one's own software gives one a different perspective than is possible from merely writing it. There is no better way to fully identify with the users of the software and get ideas for improvements than to compose with the language, perform on stage with it, and otherwise use it in the same way its users do. The more the author can do this, the better the language becomes.

A language is successful when the people behind it are committed to its success. In the end, the single most important factor in the success of a language is that its author, its users, a company, and/or an institution are committed to its continued existence. Behind every successful language, one can find an individual or individuals dedicated to the perpetuation and further development of that language. Some very interesting and beautiful languages have had to be abandoned by authors who felt compelled (economically, professionally, or simply due to their own breadth of interests) to pursue other projects.

A language cannot be successful unless it first exists. Although this may seem obvious, the first step towards creating a successful language is to start writing music software in the first place. Creative environments where many people are writing software can remove the barrier of ''over-reverence'' that sometimes surrounds existing languages (which may appear as if they had sprung fully formed out of nothing and to have existed forever). A healthy environment for language development is one in which there is a lot of software experimentation and where the conversation is just as likely to turn to software architecture as it is to the weather or the news.

A language is successful when people are ready for it. In technology, it seems that if an idea arrives even a little bit too early, it has to struggle for survival, but once the time is right, there is an explosion of development and growth. People have to be conceptually prepared to receive and understand what is presented to them, and the essential technological infrastructure must be in place in order to reach potential users and achieve the basic functionality.

A language is successful when people say that it is successful. If you live in a village where everyone believes in witchcraft and the local witch puts a public hex on you, it won't matter that you are a cheerful person in the best of health; everyone will start treating you as if you were already dead. What is said becomes the truth. And when something is in print, it carries even more authority. So having some good things said about the language in the media, on the web, in journals, and in the history books lends the language an air of credibility and permanence.

A language is sometimes successful due in part to pure luck. The successful languages are the ones whose designers were prepared to recognize and take advantage of being in the right place at the right time.

A language is successful if it has contributed ideas and stimulated new developments in the field. Even the most successful of languages will not stay around forever, but the ideas from those languages will endure and influence the direction of future developments. Human beings build languages (natural languages, computer languages, and music) as a collective project, often with the same

blind dedication displayed by termites building a nest. Each individual works, almost by instinct, on some small portion of the project without being able to fully conceive of the part it plays in the larger structure. The higher-level structure (in this case ''human knowledge'' or ''culture'') is an emergent property of the myriad of smaller structures built by individuals.

The Kyma Language

Origins

Where did Kyma come from? Kyma comes in part from a fusion of my childhood interests in both music and science and from pencil-and-paper attempts to algorithmically map patterns in nature to patterns in music written for acoustic instruments. In part it arose out of the atmosphere at the University of Illinois in the early 1980s, an environment rich in living examples of composers and engineers who were creating their own computer music languages and instruments or extending existing languages to suit their compositional requirements. (Consider, for example, Herbert Brün's elegant and algebraic SAWDUST language, Salvatore Martirano's SalMar language/instrument for real-time composition, John Melby's extensive score manipulation subroutines, Jim Beauchamp's voltage-controlled analog synthesizer and his later forays into hybrid synthesizers and computer music languages, Sever Tipei's MP1 algorithmic composition language, and the CERL Sound Group's long line of digital synthesizers and music software culminating in the Platypus user-microcodable DSP designed by Lippold Haken and Kurt Hebel in 1983.) In part it arose out of my fascination with tape music—with being able to literally hold bits of sound in my hands and to manipulate time by rearranging the pieces and taping them back together like a filmmaker. And in part it arose out of my frustration with computer music languages of the time, which felt almost like a step backward from tape music, because they were defined in terms of music notation played on ''instruments'' and could provide no real-time interaction. To my mind, though, these shortcomings were overshadowed by the promise of the structural, organizational, and algorithmic power of the computer, so Kyma was also inspired by these earlier computer music languages.

Kyma also drew inspiration from a language called Smalltalk and the paradigm of object-oriented programming. To me, ''object'' suggested something like the snippet of audiotape that I could hold in my hands. But it also suggested a more abstract conceptual grouping of atomic or compound sound objects that could be manipulated or viewed as a single entity and then ''zoomed'' to reveal arbitrary levels of detail. Kyma was also influenced by the courses I took on data structures, mathematical logic, automata theory, discrete mathematics, and graph theory at the University of Illinois. In a 1986 course on Programming Language Principles, I wrote a term paper called ''KYMA: A Computer Language for the Representation of Music'' describing an object-oriented programming language for analog circuit simulation, score representation, rule-based composition, and direct manipulation of waveforms.

I started writing code for the first version of Kyma in Apple Smalltalk running on an Apple Macintosh 512K in 1986. In 1987, I modified Kyma to make use of the Platypus signal processor as an accelerator and demonstrated it at the 1987 International Computer Music Conference in Champaign (Scaletti 1987). In 1989, Kurt Hebel and I had concluded that it was not in keeping with the mandate of a university research laboratory to build, distribute, and support hardware and software. So we formed a company, running it for the first three years from our third-floor student apartment. We assembled hardware in the kitchen, developed software in the spare bedroom, used a closet stuffed with sound-absorbing blankets as our recording studio, and used the living room for faxing, mailing, packing, and shipping. That first summer, we could not afford to run the apartment's air conditioner, but we continued programming and hardware-designing into the fall and winter, finally hearing the first sounds from the Capybara at midnight on New Years's Eve (in the first few minutes of 1990). Thanks to the extraordinary vision of our first cus-

tomers, who understood what we were doing long before anyone else took us seriously, we managed to bootstrap Symbolic Sound Corporation and move into our first real office in 1992.

Since that first version of Kyma in 1986, there have been five major software revisions, a port from the MacOS to Windows in 1992, and ports to five different hardware accelerators. The user base has expanded from a single composer/software developer working in a university research lab to an international community of musicians, sound designers, and researchers. Symbolic Sound has been, for us, an alternative means to the end of continuing our research and teaching. Our research and ideas are embodied in software, hardware, and sound, and we have been privileged to teach (and learn from) heterogeneous groups of individuals of every age and background, many of whom have also become our colleagues and friends.

In presenting this concise personal history, I hope to highlight the fact that no computer language ever springs fully formed out of nothing. Each language has a distinctive flavor to it imparted by the primordial ''soup'' in which it was spawned—a history, a personality, and a reason for being. Each language is the complex embodiment of a theory of sound, music making, and structure. The basic assumptions of each language show the imprint of the designer's personality, experiences, theories, and training.

Definition and Ramifications of the Sound Object

Kyma is a language for specifying, manipulating and combining sounds (Scaletti 1997), and it is based on the following definition:

> A *Sound* is defined to be a Sound *S*, a unary function of another Sound *f(S)*, or an *n*-ary function of two or more Sounds *f(s1, s2, . . ., sn)*.

For example, a Sound might be a source of sound (like the audio input, a sample, or a noise generator). It could be a unary function of another Sound (like a *LowPassFilter* of another Sound). It could also be an *n*-ary combination of several Sounds (like a *Mixer* with twelve input Sounds).

The definition is recursive, so one can build arbitrarily long chains of functions of other functions (for example a *LowPassFilter* of a *HighPassFilter* of a *Mixer* of three mixers and the microphone input). Some of the functions are temporal functions called *TimeOffsets* and *SetDurations*. Using these temporal functions within a structure, one can create sequences of Sounds, mixes of overlapping Sounds, and other time-varying structures.

One of the ramifications of this abstract, recursive definition is that Kyma makes no distinction between ''samples,'' ''live audio input,'' and synthetically generated signals. They are all Sounds that act as sources, and they can be manipulated and composed using the same sets of unary and n-ary functions. Another ramification of the Sound object definition is that Kyma does not draw a distinction between ''instrument'' and ''score.'' Instead, it provides an abstract way to build hierarchical structures that might or might not correspond to traditional musical organization.

This recursive definition forms the basis of Kyma. Kyma would still be Kyma even if it had no graphical user interface and no hardware accelerator (and in fact the first version of Kyma was completely text-based and ran on a single processor).

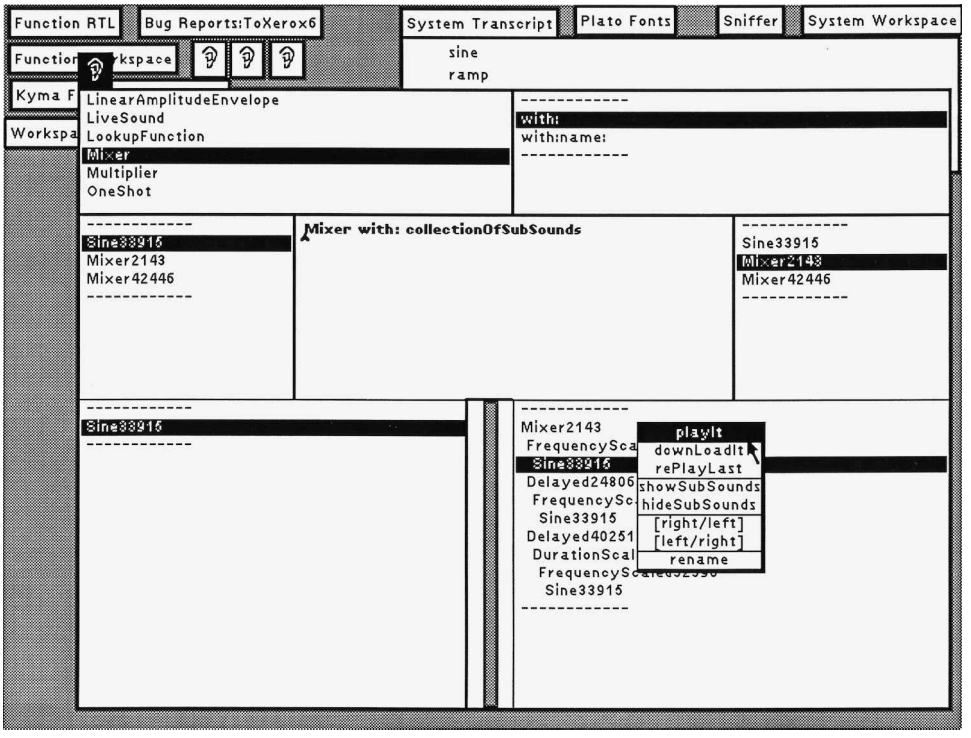
Multiple Viewpoints on the Same Abstract Data Structure

Rather than calling Kyma a ''graphical language,'' it would be more accurate to call it a language that provides multiple ways of viewing and manipulating data. The current graphical representation of Kyma Sounds has evolved over several years as I have tried to discover the most direct and appropriate representation for understanding and manipulating the Sound structures. The abstract structure came first, and the graphics evolved (and continue to evolve) in order to elucidate the structure.

Evolution of the Graphic Sound Editor

Initially, a Sound was specified symbolically as Smalltalk code. This quickly evolved into a kind of ''selection-from-lists'' interface (shown in Figure 1)

Figure 1. Kyma screenshot from 1987. After selecting a Sound class from the list in the upper left and a creation method from the list at the upper right, the creation code would be automatically pasted into the center pane. The hierarchical structure of the Sound was represented using indentation levels (lower right), and the currently selected subSound was displayed in the lower left pane. The panes to the left and right of the center pane are lists of previously created Sounds that could be used as inputs to the Sound in the center pane.



in which one could select a class and its creation method from lists, and they would then be automatically typed into a window. Previously created Sounds appeared in lists on either side of the code window, so they could be used as arguments to new Sounds. Its indentation level represented the position of a Sound in the hierarchical structure, and one could select any level and listen to it at that point.

In an attempt to further reduce the need for typing, I changed the interface to the ''Russian doll'' style interface shown in Figure 2, where Sounds are represented as ''containers'' of other Sounds. Double-clicking on a Sound opened a window showing the Sound's input(s) and parameter values, double clicking on those inputs revealed *their* inputs, and so on.

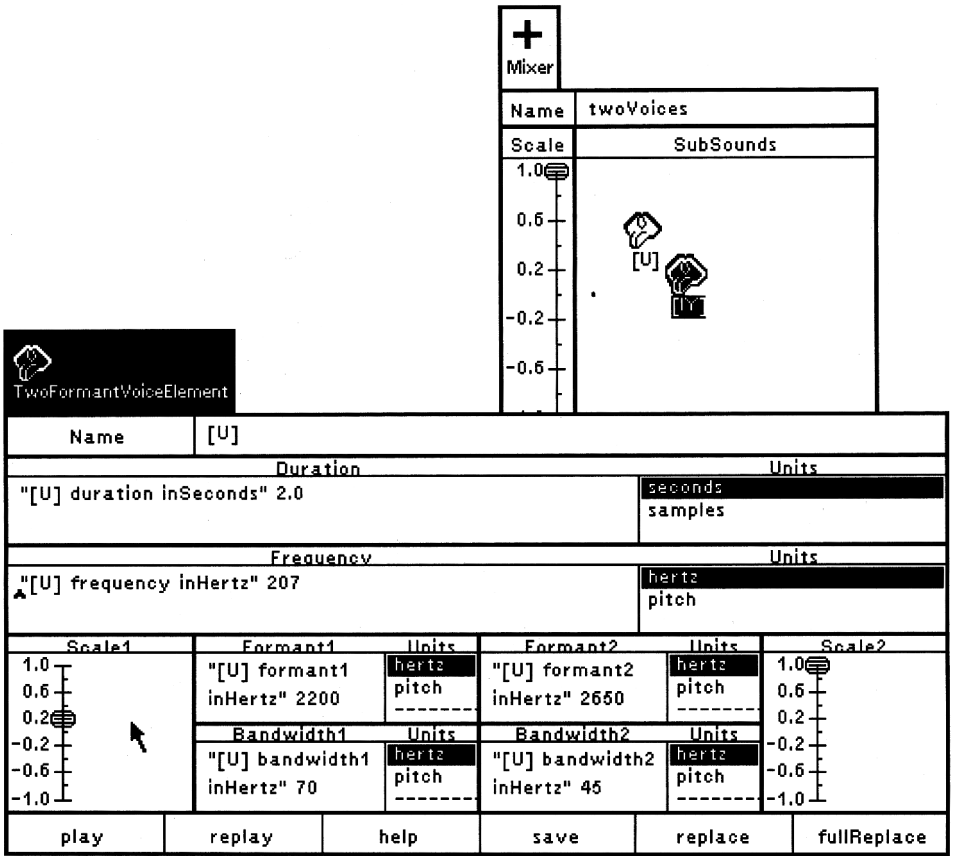
This succeeded in revealing the recursive nature of the structure but was not very good at showing the overall structure and the connections between the Sounds. Finally, I realized something that had been right at my fingertips (literally) all along. Every time I tried to describe the structure of a Kyma Sound, I always drew a picture like the one in Figure 3.

All this time, I had been describing Sounds in one way but representing them on the computer screen in a different way. It finally occurred to me that a drawing of the Sound structure could be more than just a tool for describing and understanding the Sounds: it might also be the most direct way for people to create and manipulate the Sounds. In other words, the same representation could be used for both analysis (i.e., a description after the fact) and synthesis (i.e., creating and modifying the structures). For me, at least, this was an important conceptual breakthrough. It was the first time I realized that synthesis is just one specific case of analysis.

I changed the graphical representation of Sounds from boxes-within-boxes to something that looked like the screen shot in Figure 4, a graphical repre-

Figure 2. Hierarchical Sound structure represented as boxes within boxes (Scaletti and Johnson 1988).

Figure 3. Drawing of a Kyma Sound structure (Scaletti 1989).



sentation of the functions, clearly influenced by the representation of trees and Directed Acyclic Graphs (DAGs) in computer science. The default signal flow direction was from bottom to top (an influence of the textbook illustrations of Music N unit generators).

The direction of signal flow was user-selectable, so one could reverse the signal flow direction or rotate it 90 degrees. Influenced by DSP textbooks and papers, I eventually decided to standardize the Sound representation to one showing the signal flowing from left to right (as shown later in Figure 6).

Note that through all of these changes in graphical representation, nothing about the underlying Sound structure changed. However, even something as simple as changing the direction of signal

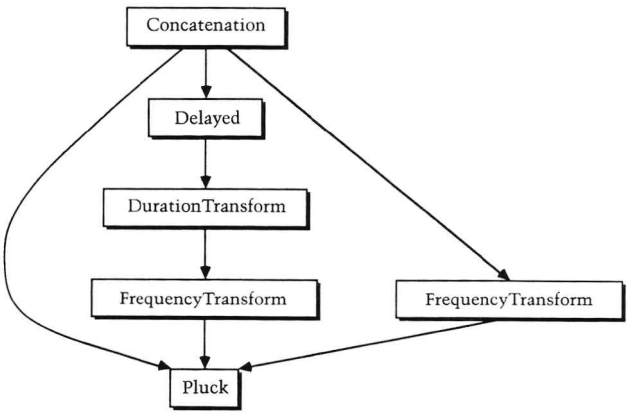
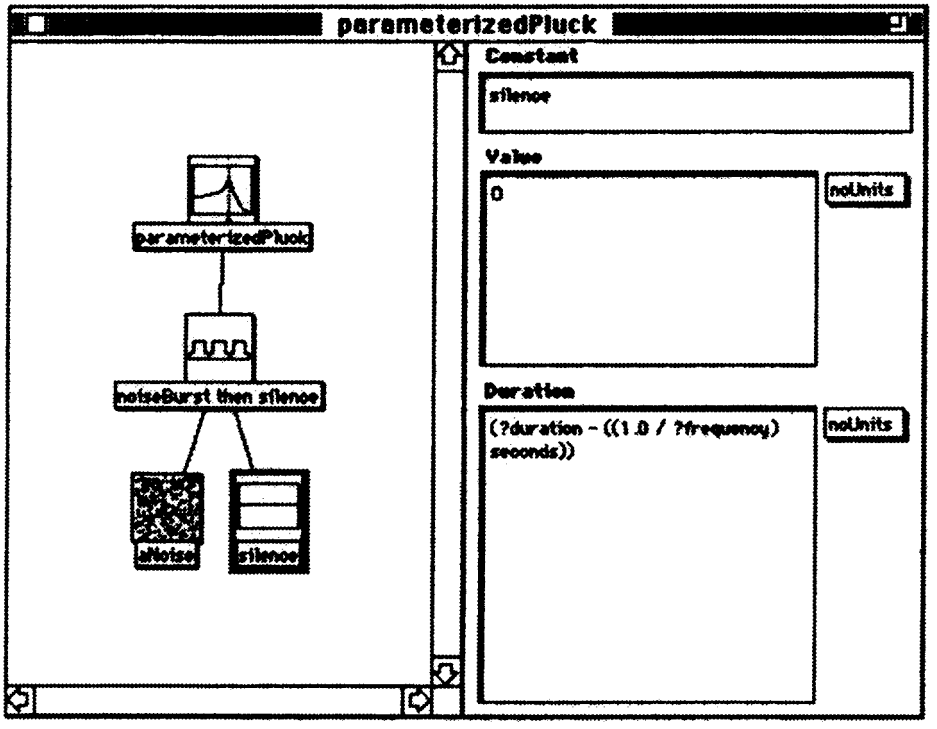


Figure 4. Early GUI version of the Kyma Sound structure (Scaletti and Hebel 1991).



flow had a noticeable effect on the way people understood and manipulated that underlying structure.

Evolution of the Graphical Timeline

Kyma's Timeline editor is yet another view of that same underlying data structure. Kyma Sounds have always been able to represent time-varying signal flow architectures through the use of the *TimeOffset* and *SetDuration* modules. These represent changes to the structure of the signal flow itself, not ''events'' in the sense of parameter updates. I noticed that whenever I wanted to explain the effects of the *TimeOffset* and *SetDuration* in a Sound (see Figure 5a), I would illustrate it using bars and vertical time markers (see Figure 5b).

Similarly, whenever I wanted to create a time-varying synthesis architecture, I would always start by sketching it out as a timeline on paper first. It was another case of finally realizing that the best representation for specifying and manipulating a structure was the one that I, along with everyone else, had already been using on paper.

I also noticed that whenever I wanted to create a parameter control function, I would first draw the desired control function on paper and then work out the arithmetic combination of functions to generate that shape. That is why the Kyma.5 timeline (shown in Figure 6) allows one to draw the control functions.

The Timeline was not so much a change to Kyma as it was the addition of an alternate view on the original underlying Sound structure. (And, judging from the number of complex, multi-layered sounds and performances that have been created using the Timeline, it must have been a closer match to the way people conceive of time varying structures).

Evaluating the Appropriateness of a Representation

At the risk of belaboring the point, all of these examples illustrate that there can be multiple ways of viewing and manipulating a single abstract structure. Each alternative can reveal or emphasize dif-

Figure 5a. Using Time-Offsets (D1 and D2) to create a time varying Sound structure (Scaletti 1992).

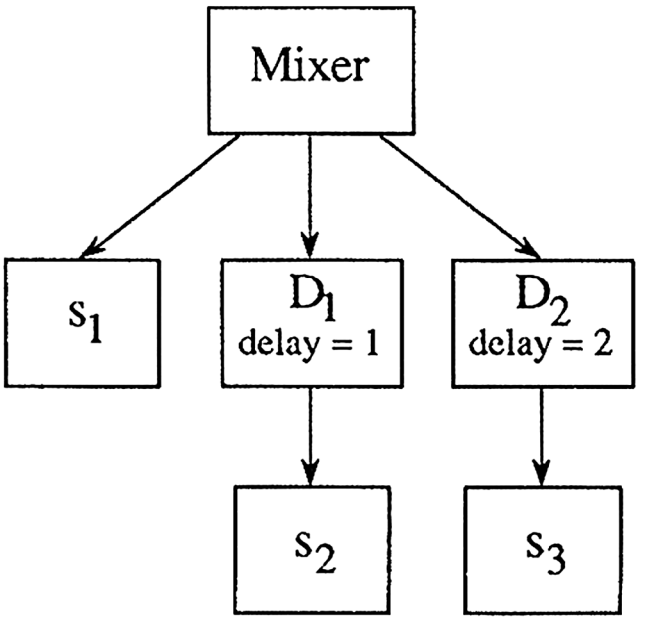
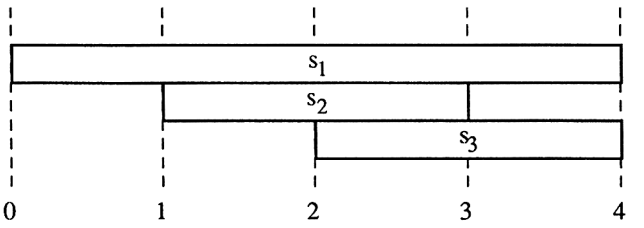


Figure 5b. Describing the results of a time-varying Sound structure on paper (Scaletti 1992).



ferent aspects of the structure and influence the kinds of things people choose to create using that structure.

Sometimes symbols are the clearest and most direct representation, and sometimes a graphical representation can be clearer and more compact. Kyma provides both graphical and symbolic interface elements, depending on the context. Arriving at the most appropriate representation nearly always requires some experimentation and a few inspired realizations. In the Kyma language, that evolution is still in progress and is part of the fun of participating in a living language.

Extending the Data Structure

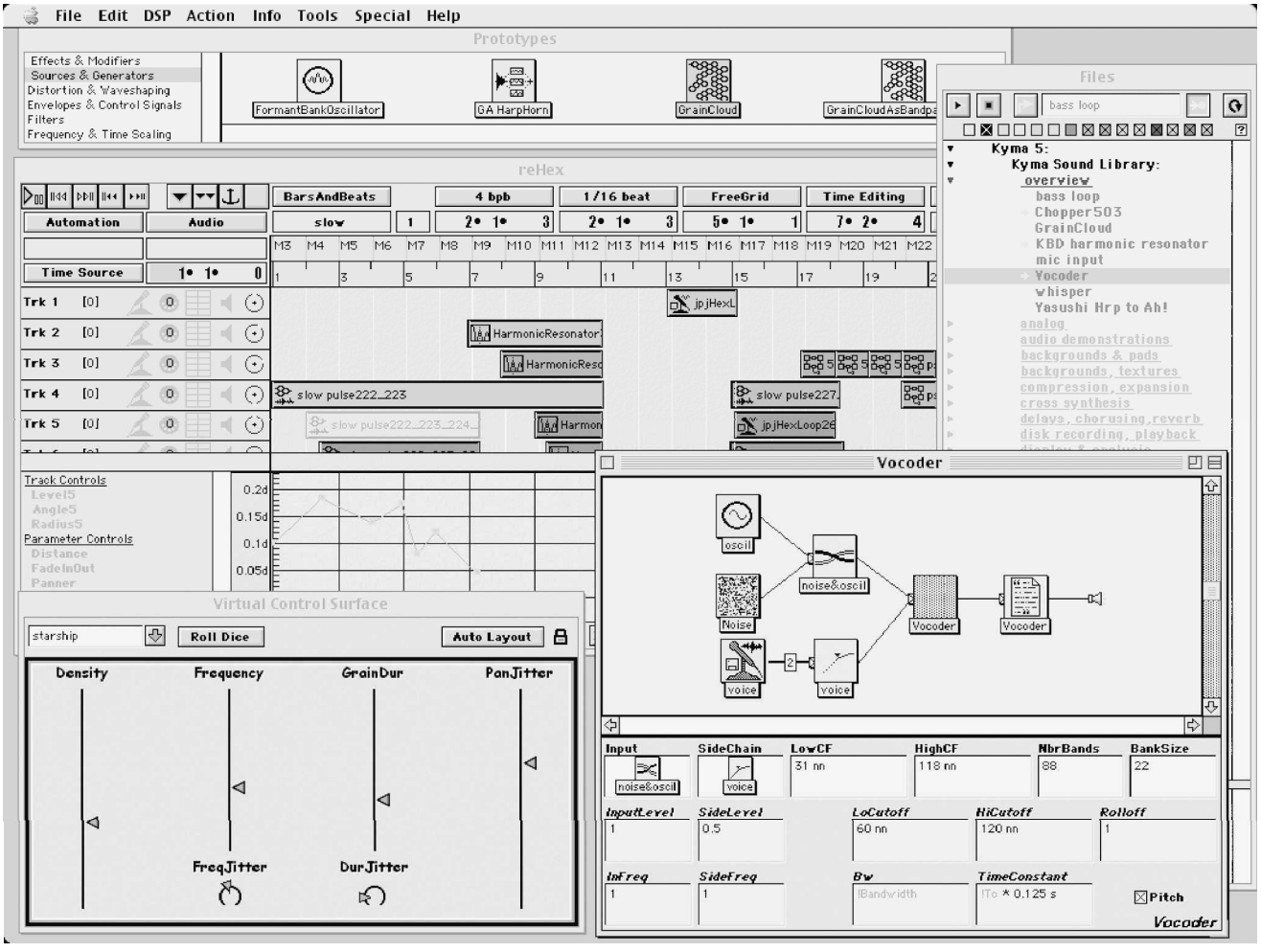
Sound objects have also proven to be extensible in ways I had not originally anticipated. In the first version of Kyma, for example, although the structures could be time-varying, the parameter values were all constants. As we and other users worked with the language, we came to realize that by making the parameters event-driven, we would not only be making the language more interactive, we would also be opening it up to external control from event sources like MIDI controllers and MIDI-generating software. So in Kyma version 4.5, we added an event language, including internal and external sources of asynchronous events and real-time expression evaluation, on the Capybara. The improvement in the quality of the sounds people made in the new version demonstrated the power of event-driven live interaction. And we were pleasantly surprised that the original data structure had been robust enough to accommodate this kind of major shift.

Expanding the Algorithm Set

One of the ideas behind the Kyma framework (and indeed behind any ''modular'' software or hardware) is that one can easily plug new sound synthesis and processing algorithms into the existing structure and immediately use them in conjunction with existing ones. Kyma has proven itself open enough to accommodate several new synthesis and processing algorithms that we have developed over the years. The most recent example is the addition in 2001 of a new family of synthesis and processing algorithms called Aggregate Synthesis.

Aggregate Synthesis is an extension to the classic ''additive synthesis'' algorithm in which complex timbres are created by adding together the outputs of hundreds of sine wave oscillators. Aggregate Synthesis extends this idea by using elements other than oscillators as the basic generators. Besides the classic ''oscillator bank,'' one can also use a bank of band-pass filters, a bank of impulse response generators, or a bank of grain cloud generators. These alternative generator banks are controlled by the same analysis file (or live spectral analysis) as the classic oscillator bank, so Aggregate Synthesis

Figure 6. Screenshot of Kyma.5 user interface (Scaletti 2000).



can be used for live analysis/resynthesis with additional control parameters like grain duration, grain envelope, grain waveform, etc.

The Role of the Capybara

Every computer music language uses hardware (because every computer music language runs on a computer). In its current implementation, Kyma runs in a distributed manner on two general-purpose computers: a Macintosh or PC and the Capybara. The Capybara is a general-purpose multiprocessor computer designed by Kurt Hebel to have a scalable architecture that, in its current incarnation, can support from 4 to as many as 28 parallel processors (with 4 or 8 analog and digital 24-bit 100 kHz audio inputs and outputs). It is not special-purpose hardware: it is not a collection of oscillator and filter circuits hardwired into a central mixer. Its function is completely determined by software. In our ''vision-centric'' society, people routinely dedicate hardware towards the acceleration and improved quality of real-time video and graphics. Audio artists deserve the same consideration (and dedicated computing cycles!).

Language as Nexus

Kyma serves as a nexus connecting sound designers, musicians, artists, programmers, and researchers who work in different geographical locations and who bring with them different sets of goals and different ways of thinking about sound and problem-solving. They use Kyma to create sounds for feature films, advertisements, and computer games; they are using Kyma on albums, in stage shows, for live theatre, in installations, in clubs, in classrooms, at parties, in laboratories, in warehouses, and at home.

I learn a great deal by talking with Kyma users, and, by inference, from watching them work, studying the sounds they create, and analyzing the questions they ask or new features they request. That knowledge becomes incorporated into Kyma and subsequently transferred, via Kyma, to all the other Kyma users who then further transform that knowledge according to their own approaches and ways of thinking about sound. Thus, Kyma is serving as a conduit for information and knowledge transfer between disciplines, resulting in some exciting cross-pollinated hybrid art forms and new approaches to processing and synthesizing sound.

Future Directions for Computer Music Software

What form will computer music languages take in a future world populated by prosthetically and genetically enhanced humans having a broader concept of art and an economy based more on services than on physical objects?

I foresee computer music languages that are perfectly scaleable, modular, dynamic networks of distributed computational elements. They will be customized aggregations of independent computational elements (some of which will be virally delivered directly into our brains) that can operate in isolation or in cooperation with additional elements accessed, only as needed, via the Internet.

I see computer music languages evolving toward more general ''virtual space'' languages in which a single database or model world drives the generation of sound, visuals, tactile feedback, physical motion, and direct neural stimulation. In these languages, it will become even clearer that the mapping is the message: given identical model worlds, the art lies in selecting or directing the observer's path through that world and in mapping the numbers into perceptual stimuli (or electrical signals suitable for directly stimulating neurons).

On the hardware side, I see an ever more urgent need for multiple, broad-band interfaces—wide channels for pumping data into and out of the virtual spaces to create a fully ''immersive'' experience.

New Humans

Hybrid Vigor

Populations are shifting and, in accordance with the principle of ''hybrid vigor,'' all this stirring up of nationalities and cultures and genes is resulting in stronger, smarter, and healthier human beings. Future computer music languages written by these future humans will embody new ways of thinking about time, space, quantity, physical movement, music, and sound.

Self-Modification

We are living through the early part of a century when life will redefine itself through genetic engineering and implant technology. There is no question that the agricultural and industrial revolutions had a big impact on human development, but the genetic revolution is more than that. It is a redefinition of what it means to be human, of what it means to be alive. We are witnesses to the moment when the DNA molecule is just figuring out how it can modify itself.

Body modifications that are now practiced as a matter of style are just rehearsals for future self-modifications that will include intelligent implants controlled by the same neural circuits as ''natural'' biological parts. The distinctions we now draw between ourselves and our machines will be blurred and ultimately erased.

New Art

Traditional distinctions between music, film, animation, and games will also become blurred. Art of the future will consist of a multi-dimensional space (literal or abstract) and a set of (deterministic or non-deterministic) ''paths'' through that space. This model is equally serviceable in describing a piece of tape music, a computer-generated animation, a live-action film, a theme-park ride, a live improvisation with computer partners, a painting, a novel, a computer game, a traditional piano sonata, a virtual environment, a live DJ mix, or an installation. ''Audiences'' of the future will consider all of these experiences to be art works and will find the old distinctions as foreign as the idea of sitting quietly in their seats during a performance.

The Space

The space of an artwork can be anything from a physical, three-dimensional space (where the dimensions would be time-stamped x, y, and z coordinates) to a completely abstract higher-dimensional space (where the dimensions might be the allowed pitch classes or timbral characteristics).

The Paths

A path through the space can be a predetermined or recorded path chosen by the artist (as in an acousmatic tape piece, a film, or a theme-park ride where the participants sit in cars pulled along tracks). Alternatively, it can be a ''live'' path (or paths) where the ''audience'' and/or improvising performers explore the space interactively.

New Economics

Software and music have much in common: both are complex abstract structures manifested as ephemeral modulations of an invisible medium. And unlike many commodities, one cannot buy music or software; one can license it for personal use, but the ownership remains with the author.

Under the old model, a musician or programmer was expected to produce a physical object (a compact disc) that could be distributed via trucks and airplanes to local storage centers (stores) where consumers could purchase them. Because consumers exchanged cash for the physical object and owned the storage medium, they were led to believe that they also owned the music or software and could therefore freely give it away or reuse any portion of it. Improvements in digital recording quality and network speeds have begun to make it more obvious that software and music are not physical objects that can be purchased and owned. Music creation, like software creation, is a process—an evolutionary, ongoing, iterative project. As such, it is ill-suited to the object distribution model that worked so well for refrigerators and bananas, for example, in the last century.

Renting the World

What if the market for music and software were based on subscriptions rather than on one-time purchases? For example, one would subscribe to a software provider, a favorite composer or author, or to a consortium of like-minded artists or programmers. In exchange for a yearly subscription fee, one would receive unlimited access to the latest software updates, the latest pieces by one's favorite composer, or even the latest work-in-progress by one's favorite artist. In the case of an artists' or a source-code cooperative, one might even be able to earn rebates on the subscription fee by contributing artwork or code to a project. (This is a variation on the open-source idea in which the most creative and productive members can contribute code and the vast majority of people who are more interested in using the product than in creating it can contribute money.)

For one thing, this would be an acknowledgement that in both music and software creation, the process is more valuable than a single snapshot or progress-report (now known as a ''release''). It is already the case that when one buys a piece of software, one does so with the implicit understanding that one will be frequently downloading or pur-

chasing updates to that software. It would be interesting to extend this concept to music as well. Instead of buying a single recording of a finished composition, one could subscribe to the composer and access updates to that composition, alternative performances of the same work, additional sections, remixes, and new variations. The best subscriptions would also include opportunities to interact with the creator in online master classes, interviews, or group discussions.

For the artists and programmers, subscription fees would provide a steadier source of income to replace the current ''feast-or-famine'' bursts of cash flow provided by the one-time purchase model. Subscribers would be like ''investors'' in a particular artist. As such, they would find it in their own best interest to help promote that artist to a wider audience and protect the artist against piracy (which, under this new model, would hardly be worth the effort in any case).

For the subscribers, it would mean better customer support, simply because an unhappy subscriber is unlikely to renew the subscription next year. Software developers already interact closely with their users and benefit from immediate feedback. Would the subscription model foster more intelligent dialogs on music as well? It could turn out that musically literate subscribers might be able to offer composers some genuinely useful observations and stimulating responses to their work (in addition to applause).

The subscription model makes it more obvious that the creator is even more valuable than that which is created. Under the old model, musicians would effectively give themselves away for free on concert tours, in interviews, bootleg recordings, and on the web in the hopes of selling physical recordings (that were owned by a record label, not the musician). That made it seem as if the compact disc were more valuable than the people who created its content (and who hold the potential for producing even more content in the future). Under the subscription model, a group of subscribers could in effect hire the musicians/programmers, investing in their future output, collectively paying the musicians's salary in exchange for wireless network access to recorded music, ''tele-performances,'' and whatever ''extras'' the artists choose to provide. (The corollary is that the subscribers could also fire the composer/programmers who were not satisfying their appetites for music and software.)

If digital content were available to subscribers wherever and whenever they wanted, then it might no longer be worth the effort to ''pirate'' or even to store music and software. If subscribers could access the content just as quickly as reading a local hard disk, and if the content might actually improve each time they access it because the musicians or programmers continue to work on it, then few people would find it desirable to steal the content.

In game theory, optimizing benefits in a one-off interaction tends to involve maximizing one's own gain at the expense of the other person. However, maintaining a longer-term relationship consisting of multiple transactions over some period of time requires some evidence of mutual benefit if both parties are to continue to cooperate. For creators and subscribers both, this new model would favor long-term, mutually beneficial relationships.

Even hardware could be sold by subscription rather than purchased outright. This would be an acknowledgment that there is more value in the design of a piece of hardware than in the actual materials used. It is also an acknowledgment of the reality that computer hardware is inevitably upgraded every few years. What if, instead of buying a computer, you subscribed to a computer hardware service? Every few years, the company would upgrade your computer to the newer model, giving the hardware companies a strong incentive to reuse existing materials as much as possible. This could provide a steadier flow of cash to hardware companies while at the same time making them more responsive to their customers, because their customers will be subscribing to a continuing service, not making a onetime purchase. It also makes explicit something that is not always so obvious in the current economy: the cost of reusing or recycling a product at the end of its useful life should be factored into the initial (or in this case, ongoing) cost of the product.

It might take some small adjustments in thinking, but in fact we are already familiar with this subscriber model in other aspects of daily commerce. It is the way we pay for news services like magazines and newspapers; it is the way we pay for communications services like telephone and Internet access; it is the way people pay for some forms of entertainment on cable television channels. In some sense, even paying taxes is subscribing to the services of the local and national government. (However, if one becomes dissatisfied with the service, canceling that particular subscription could be a bit tricky.)

Acknowledgments

Special thanks to Kurt J. Hebel, my longtime collaborator without whom Kyma and Symbolic Sound would not exist in their present form. Thanks to Jean, Paul, Matt, Barry, Perry, Robin, Mike, Donald, Desiree, Chip, and Mark who have helped run the office, ship systems, and build hardware at Symbolic Sound. Thanks to the charter members of the Kyma community: Dick Robinson, Francesco Guerra, Richard Festinger, Brian Belet, John Paul Jones, Alan Craig, Frank Tveor Nordensten, Lippold Haken, Stu Smith, David Worrall, Sal Martirano, Joran Rudi, and Bruno Liberda. And, to all of you who are now using Kyma, thank you for collaborating with us on this lifelong project!

References

Scaletti, C. 1987. "Kyma: An Object-Oriented Language for Music Composition." *Proceedings of the 1987 International Computer Music Conference.* San Francisco: Computer Music Association, pp. 49–56.

Scaletti, C. 1989. "Composing Sound Objects in Kyma." *Perspectives of New Music* (27)1:42–69.

Scaletti, C. 1992. "Polymorphic Transformations in Kyma." *Proceedings of the 1992 International Computer Music Conference.* San Francisco: International Computer Music Association, pp. 249–252.

Scaletti, C. 1997. *Kyma Sound Design Environment*. Champaign, Illinois: Symbolic Sound Corporation.

Scaletti, C. 2000. *Kyma.5 Walkthrough: A Tutorial Introduction to Kyma.5*. Champaign, Illinois: Symbolic Sound Corporation.

Scaletti, C., and K. J. Hebel. 1991. "An Object-Based Representation for Digital Audio Signals." In G. De Poli et al., eds. *Representations of Musical Signals*. Cambridge, Massachusetts: MIT Press, pp. 371–389.

Scaletti, C., and R. E. Johnson. 1988. "An Interactive Graphic Environment for Object-Oriented Music Composition and Sound Synthesis." *Proceedings of the 1988 Conference on Object-Oriented Programming Languages and Systems*. New York: ACM Press, pp. 18–26.