Blending "Traditional" Off-Line Algorithmic Textual Input with Real-Time Interaction and Graphics

Daniel Oppenheim
IBM T.J. Watson Research Center
P.O. Box 218 Yorktown Heights, NY 10598
USA
music@watson.ibm.com

Abstract

Quill [Oppenheim 1996] was added to the DMIX environment [Oppenheim 91] as a textual music input facility, modeled after the now classic languages PLA [Schottstaedt 1989] and Common Music [Taube, 1991]. The original intent was simply to make yet another compositional tool available within the DMIX environment.

However, over the past years Quill has gradually embedded itself into the DMIX system in a much more fundamental way. Thus, the traditional off-line, not real-time, algorithmic input facility now blends with real-time interactive techniques: textual input is combined with graphic displays; and meticulously worked-out algorithms developed by users in Quill work in concert with intuitive gestural and real-time improvisation.

This presentation describes the latest developments in Quill: the mechanisms for object persistency and their sharing throughout the DMIX environment, the transformations of objects into and out of Quill with the techniques of Slappability [Oppenheim 1993-4], the ability to Snap algorithms worked out in Quill onto graphic edit views of Max-like interactive objects [Puckette et al. 1988, 1990], and the Multi-Level File structure (MLF) used.

References


(This paper will be provided as addenda upon accept. (editor))

ICMC Proceedings 1994 255 Music Languages
Musically Salient Control Abstractions for Sound Synthesis

Gerhard Eckel
Institut de Recherche et Coordination Acoustique/MuSiQue, Paris, France
E-mail: eckel@ircam.fr

Ramon Gonzalez-Arroyo
Zentrum für Kunst und Medientechnologie, Karlsruhe, Germany
E-mail: arroyo@zkm.de

Abstract

We report on the theoretical guidelines and the first results of a project of research concerning the composition of an ideal environment for the composition of music using synthetic sound. The project, carried out at the Zentrum für Kunst und Medientechnologie, produced as practical result an experimental environment named Foo, which was made available to the public domain.

1. Introduction

The central issue of discussion was the domain of musical composition which integrates synthetic sound into its realm. In this context, we were focusing on the composer's conceptual approach and working methods, rather than on the aesthetic qualification of the possible results. It soon followed a particular interest in how an ideal computer environment for such purpose would look like and, therefore, we decided to build a practical system where to lay down our ideas - this would allow us to experiment with them, and later use it as a pool for further research and communication.

The project was carried out during the first quarter of the year 1993 at the Institut für Musik und Akustik of the Zentrum für Kunst und Medientechnologie Karlsruhe, and with their financial support.

Given the limited amount of time available for the realization of the project, we decided to build a model (i.e. a small image pointing to the ideal) of an environment, where the auto-imposed limitations would not hide conceptual questions. We consider the research project as a work in progress, and the already existing result as a developed statement to open discussion.

The practical result of the project is a computer package, with a generous documentation, available via anonymous ftp from ftp.zkm.de/pub.

2. Theoretical Guidelines

We want to make music with sound, and not sounds to make music with. Thus, could be expressed our point of departure. It is important for the sake of comprehensibility, however, to consider what do we exactly understand by sound, and which are the implications which for the act of composition may have the possibility of embracing the design and composition of sound matter itself.

2.1 Sound Concept

Even if sound is anything that we can hear, our interest focuses on composed sound which can be fully integrated in a compositional process. From this perspective, we define a sound concept as a musical object which describes the relationship between a musical event and its sound manifestation. It is an entity beyond physical reality, which allows for a categorization based on its audibility and recognizability, while actually implying an infinity of possible instances. The mechanisms of recognition are of extreme importance, since there cannot be any possible manipulation of an element in a logic flow, unless we can establish links between the different moments. A key word for the design of sound is behaviour, considered both at the macro and at the micro temporal domain.

From a constitutional point of view, a sound object, will in general be a complex object, decomposable into simpler elements, which we shall call components, organized under a logical structure. Thus, if undoubtedly the composition of sound is related to the signal processing algorithms, it is just as much, if not more, the description of the structure of relationships and behaviours affecting the different components which constitute it.

Therefore a sound concept can be regarded as a dynamic compound structure, where behavioural laws and signal processing configurations combine to define an object capable of being viewed both as a sound producing entity and as a logical object musically meaningful.

2.2 Compositions of Music/Sound

With the sole exception of the Analogue Studio, the composition of sound matter as such is unique to our domain, and it should logically develop compositional strategies of a different nature, as it somehow happened in the old days of the analogue technique.
We assume that the composition of sound synthesis will be affected, and will itself affect, higher logical levels of the compositional process. In fact, they can be so variedly intermingled that there is no way to set a precise boundary between the level of sound object definition and that of its musical manipulation. This has an important implication: we cannot assume one paradigm of conceptualization, such as the most popular conceptual distinction between instrument and score. However, we are not proposing a tabula rasa view where all levels are mixed up, but rather assuming an interest in a search for other perspectives of relationship between sound matter, musical material and form. The tools to build conceptual levels and abstractions should, therefore, be these.

2.3 Analysis & Research Areas

We can conclude from the above, that we wish to see embodied in one whole all actions from the micro-control of the signal processing modules to the composition of the score. A first implication is that two domains with very different requirements will coexist in one sole environment: The Signal-Processing Domain -with its strong demand for computational power- and the Symbolic Processing Domain (constituted of as many logical layers with their abstraction and representational tools as needed for the composition of a piece of music), urging for multiple perspectives and flexibility in expression. Although a discussion escapes the scope of this article, it can be shown that the ways in which these two domains should communicate are multiple and all-directional. Our Signal-Processing Domain should, therefore, feature a real flexible language.

The treatment of time in both domains also differ widely. While the one-dimensional time flow of the Signal Processing Domain advances in small units which we want to maintain for qualitative reasons; time in the symbolic domain is a main subject of its processing, and one cannot talk any longer of a prevailing uni-directional flow. Certainly, these two represent extremes in what we could consider a hierarchical structure of time perspectives. Similarly, we can postulate the existence of an equivalent structure of logical levels. These two hierarchical structures will have to have multiple inter-connections, but cannot be assumed to follow a strict parallel correspondence: a step higher in the logical hierarchy does not necessarily imply a step higher in the time hierarchy, and vice versa.

A last point to mention refers to the creation of abstract musical objects and their functional symbolic representation. We can understand the representation of an object as the creation of a perspective over its behavior (i.e. its manifestation in a certain context), which focuses on certain aspects related to the potentialities of development to be exploited in a certain musical situation. Identical objects, from a constitutional point of view, could have extremely different representations, depending on the features which want to be underlined and on the way they might be influenced by the context in which they will be evaluated.

3 Foo Environment

The Foo environment consists of two parts: the Foo Kernel layer and the Foo Control layer. The Foo Kernel layer is implemented in Objective-C and its functionalities are made accessible to Scheme through a set of types and primitives added to the Elk Scheme interpreter [Laszlo and Norressen, 1993]. The Foo Control layer is implemented in Scheme and OOPS, an object-oriented extension to Scheme which is part of the Elk distribution. Whereas the Foo Kernel layer implements the generic sound synthesis and processing modules as well as a pitch description and execution language, the Foo Control layer offers a symbolic interface to the kernel and implements musically salient control abstractions. The user interacts with the Foo environment by writing Scheme programs which eventually will define and execute sound synthesis patches in non-real-time.

3.1 Foo Kernel

The Foo Kernel layer provides the necessary low-level abstractions to define and execute signal oscillation processing patches. Construction and compilation of patches is made efficient in order to allow for the definition of a new patch whenever desirable. The normal interaction with Foo is to create a patch for every self-contained event and test - as in more traditional approaches - to predefine patches which are then instantiated, initialized with parameters, and executed. Although the structure of a patch cannot be changed during execution Foo provides for the need of dynamically changing parameter generating functions (e.g. to realize a filter bank with a variable number of filter elements). Thus the Foo Kernel invites to write generic patch generation code - a feature the Foo Control layer massively relies on.

3.1.1 Kernel Abstractions

For conceptual cleanness and orthogonal implementation the Foo Kernel provides three abstractions useful to represent patches: modules, signals, and substrates.

- Modules represent signal processing algorithms which produce signals and may consume signals or read substrates (e.g. generation, sample readers).
- Signals represent streams of sound samples. They are used to represent sound in-time. The connections between modules in a patch are represented by signals.
- Substrates represent sequences of sound samples or control functions out-of-time (e.g. sound files,

IKMC Proceedings 1994 257 Music Representation, Data Structures
break-point functions). Substrates can be read by modules in order to convert them into signals.

The Foo Kernel provides two other abstractions to create and execute patches: contexts and tasks.

- A context contains all modules, signals, and substrates that are to be executed together. There is no communication between contexts. The context contains all information about the temporal relationship between the modules. Contexts can be stored on disk in a compressed binary format for later compilation and execution.

This feature is the basis for the incremental mixing support discussed below.

- A task is an execution environment for a patch defined in a context. It differs from the patch execution sampling rate, the processing block size and the sound output medium. The task is built from a context by an optimizing compilation process.

3.1.2 Time

One of the differences between the Foo Kernel and other synthesis languages is the possibility to express the temporal relationship between different modules on the level of the patch. Therefore the Foo Kernel defines modules which are sensitive to the temporal context in which they are employed. A syntactic extension to Scheme allows to define such temporal contexts which implement what we call static temporal module binding: the time origin of a time-sensitive module is located on the global time line at a position specified by its local and enclosing temporal contexts. This enables the user to easily represent parametrizable hierarchical time structures.

3.1.3 Execution and Optimization

A design goal of the Foo Kernel development was to hide all complexity of the patch execution from the user. We were seeking a patch description language as declarative as possible. This was accomplished by allowing the user to describe the patch without being concerned with efficiency issues at all and by optimizing the patch before and during its execution. Before execution the Foo Kernel detects all redundant operations in the patch (e.g., multiplication by a constant signal with value one) and removes them. During the execution only the currently active time-sensitive modules are executed. All time sensitive modules can determine the periods during which they have to be active and if a module is inactive it is assumed to produce a signal with value zero. This information is used by the Foo Kernel scheduler for all dynamic optimizations (e.g., a whole sub-patch whose result is multiplied by zero does not need to be executed at all).

3.1.4 Incremental Mixing

By incremental mixing we understand the possibility to mix the signal resulting from the execution of a synthesis patch into an existing sound file. The Foo Kernel supports this procedure to the extent that it allows to remove a layer that was mixed in incrementally. This feature is based on the property of the Foo Kernel Context to completely represent the result it may produce upon execution. In incremental mixing mode the Foo Kernel stores each context used to produce a layer in the incremental mixing file structure. This allows to reuse the context again at a later time and to mix in it with reverse sign which results into a complete cancellation of that layer.

3.2 Control Abstractions

The Foo Control layer provides control abstractions for the creation, representation, and manipulation of sound concepts and other musical objects in general. The abstractions are accessible to the user in form of a coherent set of Scheme types and functions or OOPS classes and methods. The programming tools provided by the Foo Control layer provide powerful means for the automatic construction of optimized Foo Kernel patches, as well as for their integration in a symbolic compositional layer.

3.2.1 Kernel Interface

An important part of the Foo Control layer is its interface to the kernel. It connects the signal processing domain with the domain of symbolic processing. The concept of signal processing component, which may represent any generic kernel module, permits a polymorphic description of the lowest-level input parameter as control functions with their specific temporal evolution and modulation or the modalities of accessing soundfiles). An abstracted control entity allows to recursively define parallel and sequential groups of signal processing components (e.g., banks of oscillators or cascades of filters). Each signal processing entity provides at its input and output a signal routing matrix for dynamic dispensing. Besides the possibility to explicitly set treat time offsets (based on the functionalities provided by the Foo Kernel) the Kernel Interface library provides for global duration and amplitude scaling. The basic purpose of the Kernel Interface is to offer powerful access to the kernel features through a generic and compact notation. This is the first step towards an integration of signal processing functionalities with the symbolic world of music composition.

3.2.2 Node

The Node is an abstraction that integrates different ways to generate and transform control objects of any kind. It provides various strategies to structure and relate collections of objects. Streams of values can be produced by various generators selectable
3.2.3 Envelope

Based on the break-point function concept, the Envelope abstraction was conceived to describe the temporal evolution of any kind of control parameter. What distinguishes the Foo Envelope from more traditional approaches is the possibility to describe rather classes than single instances of parameter trajectories, as they are needed to represent sound concepts. Foo Envelope can be used to relate along a monotonously increasing dimension objects of any kind and not just numeric quantities. This feature accounts for the usefulness of this abstraction at almost all levels of control. A Foo envelope represents a partially instantiated description of a control trajectory which receives a final form in the context in which it is evaluated. In practice, the two most important context parameters are the time origin and the duration factor. It contains a resolution mechanism that computes the final form of the envelope from its generic description. A small but powerful envelope description language allows to express all desirable temporal relationships between the different break-points, which are actually Foe Nodes. A Node's position can be described relative or in any other Node in the sequence or in absolute units of the enclosing time context. For each Node a relative or absolute minimum and maximum value can be declared to be respected by the constraint engine. Besides this flexible description of the position of the break-points the Envelope abstraction also offers a second functionality: the user may specify a function for each node describing the relationship between the parameters within the corresponding interval.

3.2.4 Abstraction

The most general of all control modules is called Abstraction. As the name clearly illustrates, its purpose is to serve as a general tool to create abstractions of families of them, by offering a kind of meta-abstraction mechanism. The Abstraction class provides a template of an execution context, in which a set of computation mechanisms may act upon a number of "components" (abstraction: themselves) some local data and data provided by the context in which it is situated. This very open and general structure is complemented by the possibility to define a hierarchy of composition to be respected upon execution. In general we consider the Abstraction class the basic building block for the definition of user-defined control paradigms.

3.2.5 Processes

Foo Processes offer yet another way of dealing with time (besides the kernel time construct, time sensitive Kernel modules, and the notion of time in the Envelope). Inspired by the concept of process in Formes (Rodet and Coine, 1984) the Foo Process abstraction was designed to allow the description of a priori independent processes which evolve concurrently in time but may communicate with each other. The Process abstraction consists of two classes: Process and Scheduler. Processes are registered with a Scheduler which sequences their execution and communication. After a Process was run it may decide when to be run again, which allows to implement temporal control with changing granularity. A process can be desactivated until another one activates it again. Because of its orthogonality with respect to the other control abstractions the Process abstraction may be used on all levels of synthesis control.

4 Related Work

We would like to mention here the different sound synthesis and computer aided composition systems that inspired our research: Project Two (Koenig, 1970), Chaos/Formes (Rodet and Coine, 1984), Midim/Voyim [Kagel, 1986], Cmusic [Moore, 1990], Common Lisp Music [Scheiritsdoldt, 1991], Common Music/Stella [Tatei, 1991], Csound [Verge, 1991], and Vqsynth [Dannenberg, 1993].

References


References


