Control Mechanisms in the MOSAIC Synthesis Program

Joseph Derek Morisson
IRCAM, 31 rue Saint-Merri, Paris, France
(currently at: GTE Laboratories, 40 Sylvan Rd, Walthem, MA, USA)
(617) 466-3614
Email: jmorisson@gte.com

Jean-Marie Adrien
IRCAM, 31 rue Saint-Merri, Paris, France
(01)33-1-42.77.12.33
Email: adrien@ircam.ircam.fr

Abstract

MOSAIC is a synthesis program based on physical models. Sound is generated by instructing MOSAIC to move objects along trajectories, causing them to strike or rub other objects, etc. The sounds produced by MOSAIC are very natural and "acousticounding", but the synthesis is very difficult to control, since one must specify to the computer the physical, low-level details of how to play an instrument. To alleviate this difficulty, MOSAIC provides both continuous and discrete control mechanisms for controlling the physical models. Continuous control can be used to specify, say, the position of a violin bow over time, but discrete control is more appropriate for high-level commands such as "play middle C for 1 second, forte". MOSAIC contains mechanisms for converting back and forth between continuous and discrete control, for scheduling complex tasks over time, and for creating "feedback" systems where the control can respond dynamically to the synthesis. This is all based on an interpreter for the Scheme programming language, therefore users can specify arbitrarily complex rules for how discrete commands should map to continuous control parameters. MOSAIC also contains a MIDI file parser, and can therefore make use of data generated by MIDI consoles or patch programs such as MAX. Currently, several composers at IRCAM are experimenting with MOSAIC. Work is in progress on a graphical interface, and on making the software run in real time on the IRCAM workstations.

Introduction

MOSAIC (Modal Synthesis and Analysis with Interactive Control) is a synthesis program based on physical models. The modeling algorithms in MOSAIC were developed by Jean-Marie Adrien [Adrien, 1988].

In MOSAIC, an instrument is represented as a collection of mechanical or acoustic resonant structures, vibrating and interacting together. The resonant structures are used to represent violin bridges or bodies, acoustic tubes, bells, timpani membranes, strings and so on. Any number of access points can be created on each resonant structure. Resonant structures can be coupled by specifying connections to be made between pairs of access points. Connection types include adhere, strike, bow, pluck, and so on. Sound output is obtained by designating one or more access points as virtual pickups.

MOSAIC is based on an interpreter for the Scheme programming language [Abelson and Sussman, 1985], [Rees, Clinger, et al., 1986], thus the MOSAIC language is a superset of Scheme. For the interpreter, we used Oliver Laumann's excellent ELK Scheme implementation [Laumann, 1990].

The MOSAIC execution model

MOSAIC consists of a Scheme interpreter augmented with several datatypes and primitives, and a MOSAIC workspace in which the user constructs a machine which simulates the physically-modelled instrument. The user interacts solely with the Scheme interpreter, so there are primitives available in the interpreter to add components to the machine, to destroy the machine and to run the machine.
An example: synthesizing a plucked string

Here is the MOSAIC code one would write to synthesize a plucked string:

```mosaic
;\[ Mosaic 3.0\] plucked string example
(new) ; destroy the old machine, reset time to zero

;\[ make objects (use default values for physical parameters\]
(define my-string (make-object 'non-string))
(define my-plectrum (make-object 'bi-two-mass))

;\[ make pluck connection\]
(define my-string-pik (make-access my-string (const .6) 'trans0))
(define my-plectrum-pik (make-access my-plectrum (const 1.0) 'trans0))
(make-connection 'pluck my-string-pik my-plectrum-pik 0.1 (const 50))

;\[ make position connection to push plectrum\]
(define my-plectrum-mot (make-access my-plectrum (const 0) 'trans0))
(make-connection 'position my-plectrum-mot (make-controller 'envelope 1
((list 1.00 .1)
 (list 0.50 -.5))))

;\[ make listening point on string\]
(define my-string-out (make-access my-string (const .3) 'trans0))
(make-point-out my-string-out)

;\[ the string-plucking machine is now built\]
(run 2) ; run the machine to generate 2 seconds of sound
(play) ; play the sound
```

A full explanation of this example is beyond the scope of this paper. However, the important point here is to understand the role of the controllers. A controller is used whenever a (possibly) time-varying value must be specified in the machine. The (const ...) function creates a constant-valued controller, and the (make-controller 'envelope ...) function creates a breakpoint envelope controller. A good way of thinking about it is to imagine that the MOSAIC machine is a black box with some knobs on it. There is one knob on the black box for each controller. When the black box is "run," the knobs control the horizontal and vertical positions of the plectrum, etc. Plucks are heard if the vertical position knob is moved. Any knob corresponding to a constant controller is "glued" in place, and can never be moved. Any knob corresponding to a breakpoint envelope controller automatically follows the envelope when the machine is run.

MOSAIC also provides more sophisticated, dynamic controller types than the above which provide the basis for the features described in the abstract. The rest of this paper describes these controller types.

Controllers as input to other controllers

Controllers can use other controllers as input. The sine controller, for instance, takes another controller as an input to specify its frequency. This means that one can do simple FM synthesis using only controllers!

Dynamic controllers (discrete to continuous control)

A dynamic controller is similar to a constant controller except that after it has been created, one can send it (time, value) messages telling it to change its value to the specified value over the specified amount of synthesis time. Here is a code fragment using dynamic controllers:

```mosaic
;\[ make dynamic controller, initial value 2.3, updated every .001 synthesis seconds\]
(define dc (make-controller 'dynamic 2.001 (list 2.3)))
```
... include some Mosaic code which uses do as a controller ...

(run 2.0)
(set-breakpoint dc (list 2.0 3.71)) ; ask dc to change to 3.7 over the next 2 seconds
(run 2.0) ; do changes during this run

Scheme controllers

A Scheme controller periodically runs a Scheme function to decide what its value should be. The function in a Scheme controller actually runs while the machine is running. Functions defined in a (make-controller 'scheme ...) expression can refer lexically to variables in the enclosing code (i.e., Scheme controller functions can take advantage of Scheme's lexical scoping). An argument in the (make-controller 'scheme ...) expression specifies how often the Scheme controller should be called to update its output value. Normally, Scheme controllers are updated relatively infrequently (every few hundred samples) to compensate for the fact that invoking a Scheme function during a synthesis is expensive.

Example: simple control value transformation

Here is how to create a Scheme controller whose output is the sum of two other controllers. Its output is updated every .01 "synthesis seconds", i.e., every 441 samples if the sampling rate is 44.1 kHz.

(define cl (make-controller ...))
(define c2 (make-controller ...))

(define sum (make-controller 'scheme 1 .01
(lambda ()
(vset 0 (+ (vget 0 cl) (vget 1 c2))) ; add the 2 inputs
2 (list cl c2))))

Example: continuous to discrete control

This example involves a Scheme controller whose output value is not used. Every time the value of its input controller crosses zero, it calls the quux function.

(define cl (make-controller ...))

(let ((was-positive (> (get-info 'value cl U) 0)))
(make-controller 'scheme 0 .01 ; wake up and check things every .01 synthesis secs
(lambda ()
(let ((is-positive (> (vget 0 cl) 0)))
(if (not (eq? was-positive is-positive))

(qwqin (set! was-positive is-positive) (quux))))))

Example: scheduling tasks over time

The simplest way to create a smooth control in Mosaic is to have a Scheme controller periodically reset the breakpoint of a dynamic controller:

;;; initial value 0, update every .001 synth secs
(define ramp (make-controller 'dynamic 1.001 (list 0))

;;; ramp period: 2 secs, ramp maximum value: 3.16
(make-controller 'scheme 0 2.0 ; run function every 2 synthesis seconds
(lambda ()
(set-breakpoint ramp (list 0.0 0.00)) ; set value to 0
(set-breakpoint ramp (list 2.0 3.14)) ; set up next ramp
0 '())))

ICMC 21
Note that in general we would want a ramp with variable frequency; this would require a more complicated scheme controller with a much higher trigger rate (which would only occasionally reset the breakpoint).

Many scheduling tasks can be handled with this sort of Scheme-controller-at-clock technique. At IRCAM, we created a flexible timer mechanism and a score-management mechanism based on this technique.

Feedback systems

The MOSAIC get-info function returns information about the dynamic state of the current synthesis. A Scheme controller can use this information to decide its own output and can therefore "react" to the machine's dynamic behavior. For instance, at IRCAM we created a "smart hammer" which used a Scheme controller to specify the force on the hammer's handle. The Scheme controller would "push" on the handle until the hammer struck something; then the Scheme controller would pull the hammer away.

MOSAIC assemblies

The mechanisms discussed above combine well with Scheme's excellent abstraction facilities. For example, one can create a synthesis of a plucked string, then enclose it in a (define (make-pluck-string-assembly ...) ...) expression. It can then be parameterized and reused in a variety of subsequent syntheses. One can even have the note-pluck-string-assembly function return another function which can in turn be invoked with various arguments to send messages to parts of the assembly. (This is essentially object-oriented programming, without inheritance.)

At IRCAM, we have made some fairly complex assemblies, and we used assemblies as components of larger assemblies. We hope that some of the techniques musicians use to play real instruments can eventually be formalized and "captured" in these MOSAIC assemblies.

MIDI and real time

MOSAIC contains a midifile controller type which extracts user-requested information from a MIDI file and outputs it as controller values. Scheme controllers can use midifile controllers as input controllers, and can therefore initiate actions based on MIDI events. MOSAIC synthesis designed to use midifile controllers will automatically work properly with real MIDI, when MOSAIC runs in real time. Thus, composers can work on "instrument performance" algorithms without having to commit to real time or batch operation.

Conclusion

MOSAIC is implemented in C++ and runs on NeXT machines and DEC workstations (computation time is between 10 and 1000 times real time, depending on the machine and the complexity of the synthesis).

Currently, several composers at IRCAM are experimenting with MOSAIC. Work is in progress on a graphical interface, and on making the software run in real time on the IRCAM workstation.

References


