DESIGNING AN AUDIO INTERACTION MODELING LANGUAGE: SOME BASIC CONCEPTS AND TECHNIQUES

Ioannis Zannos
Staatliches Institut für Musikforschung
Tiergartenstr. 1,
D-10785 Berlin,
Germany.

email: iani@sim.spk-berlin.de

Abstract

This paper discusses issues in the design of scripting languages for interactive audio-visual applications. Starting with the concept of *event*, it extends its property of "onset time" through the feature of *trigger condition*, to allow embedding of interactive elements in partially fixed temporal structures. An object-oriented implementation gives rise to a versatile class which can serve as basis for interactive scripting tools. Furthermore the feature of dynamic connections between event-process instances is introduced, thereby allowing communication in a dataflow manner as encountered in "MAX", but with all the advantages of a full programing language.

1 Events and Sound Objects

The concept of "*event*" is fundamental for computer based real-time systems, since it deals with the basic features that are necessary for performing timed actions in such systems. It has been discussed in relevant literature, notably by Pope (1997: 331f). Roads (1996:695) regards duration as a fundamental component of event[1]; but MIDI-events have no duration and so have many other objects in other systems. Some other terms used with almost identical but not clearly delineated meanings in the literature are *sound object* (Pope 1997, Dannenberg, Desain and Honing 1997:309f), *sound event* and *musical object*, furthermore *sound instance* (Dannenberg, Desain and Honing 1997:277). Related to these is a similar group of concepts dealing with groups of events as for example: *event set* (Dannenberg, Desain and Honing 1997:277), *event list, timed event list, event stream* (Dannenberg, Desain and Honing 1997:310). Here we take a closer look at the content, duration and timing aspects of events, in order to clarify the semantic field of this term in a way that is both compatible with common usage in systems like MODE, GTF, Nyquist, described in the literature cited above and that can serve as basis for definition and implementation of more complex objects involving interaction. We start with a simple definition of event:

An event is something that happens during a performance. For example:

- A single sound
- A change in a setting, that changes the audible qualities of subsequent sounds (change in loudness, timbre, etc.).
- A gesture that causes a group of sounds or other events to start or to stop playing.

In computer music systems events can be:

- MIDI-Messages
- Function calls or evaluations of function closures (for example: foo.value(x, y, z);
- Evaluation of a messages sent to an object (a class instance)

Events may or may not have a duration. Events that have a duration are usually audible tones, pauses, groups of tones, or otherwise sections (parts) of the performance. Events without duration are often "triggers", that is signals indicating that something (an event, a group of events) should happen, like for example the beginning or end of (groups of) other events, or other changes in the flow of a performance. It is proposed here to use the term "*sound object*" to distinguish objects which have duration from events without duration.

1. A sound event is "an object that has a duration and other properties". (Roads, 1996:695)

2 Event Timing: “Time Offset”, “Trigger Condition”, “Time Point”

2.1 Trigger Conditions

In traditional music performance from a score, the order and to a certain extent the exact timing of events is predefined (determined). A common way of defining the timing of events in computer music systems is to provide each event *e* with an offset which states the time that elapses between a previous event *e*- or some “outer scope” and e[1]. While time offset is almost always necessary to define the time structure of music performances, it is not the only useful concept for this task. Here we propose to add the concept of *trigger condition*, along to that of time offset as a way of determining the timing of events. This is motivated by two facts:

- Some events may happen only as immediate response to external input, without any time offset
- The trigger condition concept is useful for the implementation of interactive systems, because its affinity to the idea of checking a rule (condition) and of triggering an action by a function call or a message to an object.

In interactive music systems like the one we deal with here, many events depend on input from the environment, that is, human performers as well as possibly other persons or things interacting with the system. This is also comparable to the way musical improvisation works. One can therefore say that in general events are “triggered”, caused to happen, at certain time points during the performance. Triggers are conditions which, when they become true, cause an event to occur. Dannenberg, Desain and Honing (1997:308) point out that rules of the form “if condition then action” are used in rule-based systems to obtain an extremely flexible information flow. It is this flexibility that speaks in favor of defining triggers as conditions, not only within rule-based systems, but for interactive music systems in general. Examples of trigger-conditions are:

- Input of a certain item from the user via a devices such as the mouse, computer keyboard, MIDI-keyboard or movement tracking device. We could formalize this as: input(<message> <device>) == true. meaning: it is true that <message> has been input over <device>.
- Receiving a certain message from another system, for example from a program operated by another user, via TCP/IP. We could formalize this as input(<message> <port>) == true. meaning: it is true that <message> has been input over <port>.
- The detection of a certain feature in the input, for example, “an octave step has occurred in the melody of the input” or of some internal condition computed by the system on the basis of past input and its processing.

2.2 Untimed Events and Timed Events, Time Offset

If an event is supposed to happen immediately when its trigger condition is fulfilled, then we call it an untimed event. When on the other hand, an event is supposed to happen a specific time interval after the trigger has occurred, then we talk of a timed event. We may call the interval between the condition (which constitutes itself an event) and the onset of the event itself a time offset. In most systems musical events are always associated with a time offset. On the other hand, it seems fitting to define events that happen entirely as immediate response to input from the user or the environment as untimed events in the sense given above.

2.3 Time Point

Time points can be also regarded as abstract entities separate from the contents of their events. According to the above, a time point is defined by:

1. the condition c which triggers it (causes it to be defined within time) and
2. optionally, by the time interval that elapses between the occurrence of the trigger and the occurrence of the time point.

3 Implementation Example: The *Poller* Class

As an example the realization of the above concepts in an object oriented framework is described here through a class called *poller*. Poller is capable of calling repeatedly a function, which can be configured to read values from single objects or from groups of objects in the system, compute a result and/or execute an action (including the playing of a sound) and send the values obtained (or generated/calculated) selectively to single objects or to groups of objects in the system. We call the repeated call of this function “polling process”. A Poller can only run one polling process at a time.

1. See Roads (1996:695): “the start time of an event is always defined relative to some outer scope”

The polling process can be started and stopped by sending Poller instances the messages 'start' and 'stop'.

We call the function that obtains or computes the data a “polling function”. This is stored in variable '*pollfunc*'. Since *pollfunc* can be set freely to any function or object at any moment during a Pollers activity, the configuration of the polling process is very flexible. Possible tasks of the polling function include the processing of data obtained through algorithms of any degree of complexity, and the generation of data by SuperCollider Stream objects (SList, Sseq, Sser etc.) or by pollable generators such as Pline, Poscili, Psinosc etc. Several methods are provided by Poller or its subclasses for setting and configuring polling functions to obtain different kinds of behaviors.

Furthermore, the duration between subsequent calls of the polling functions can change at any moment, by changing the value of the instance variable '*rate*' or by returning different values as a result of the evaluation of a function stored in the '*rate*' variable. In this way, the poller can change tempo, or play practically any kind of algorithmically or interactively definable rhythm.

A list of “*receivers*” is provided for communication with other objects. The current value list computed by the polling action is used as parameter for evaluating all functions stored in a list '*receiverfuncs*'. The functions added to the variable *receiverfuncs* are identified by a tag stored in the associated list 'receivers', so that they can be removed later by reference to this tag. A number of default message-sending template creators are provided for connecting other objects as receivers to the poller. In this way, complete networks in the manner known from MAX can be easily configured. The difference is, that whereas in MAX connections are mostly fixed during the entire execution of a program, here connections can be freely added to or deleted from an object at any time during performance. This gives the potential for creating truly dynamic, evolving networks of performing artificial “agents”.

The start method of Poller is responsible for starting the polling process. It does this by wrapping a shell around the polling function which handles the tasks of evaluating the polling function stored in instance variable, 'pollfunc', storing its result in instance variable 'currentval', distributing it to the receivers stored in receiverfuncs, and rescheduling the polling process to be called again after "rate.(currentval)" seconds. The structure of the polling process cannot be changed within one class, but subclasses of poller may override the start method to provide different types of polling processes.

4 Implemented Features Example

A selection from the method library implementing timed and conditional starting and stopping of the polling process and its connection to other pollers is presented here. Its purpose is to show how the start and stop methods combined with the basic framework of poller (*counter* and *rate* variables) and the idea of connecting to other *Poller* objects created dynamically lead to concise and clear formulation for a rich variety of combinations of triggered and timed execution. The examples are coded in SuperCollider[1], version 1.8.

```
(* Setting the poller to start / stop at some future time point. start running after "offset" seconds. Options according to
offsets sign: If offset < 0: dont start, 0: start now: > 0, start after offset seconds *)
method startAt { arg offset;
	if offset > 0 then [offset, { this.start }].sched; else if offset == 0 then this.start; end.if; end.if; }
-- start running now, stop after "dur" seconds.
method runFor { arg dur; this.start; [dur, { this.stop }].sched; }
-- start running after "offset" seconds, run for dur seconds after that.
method startAtFor { arg offset dur; [offset, { this.start }].sched; [dur + offset, { this.stop }].sched; }
-- start and keep running until 'acondition.(currentval)' becomes true. offset option: see method startAt.
method runTill { arg acondition, offset = 0;
	this.addReceiver('stopCondition', { arg cval;
		if acondition.(cval) then this.stop; this.removereceiver('stopCondition') end.if; });
	this.startAt(offset); }
-- run for 'times' times, then stop. offset option: see method startAt.
method runTimes { arg times = 1, offset = 0;
	this.addReceiver('counter', {
		count = count + 1;
		if count == times then this.stop; this.removereceiver('counter'); this.reset; end.if; });
	this.startAt(offset); }
-- Perform anAction when 'cond.(curval)' becomes true in aPoller.
```

1. For info on this real-time audio programming system see: http://www.audiosynth.com

```
-- Basis for startIf, stopIf etc.  Options: startflag != 0: start aPoller now.
-- stopflag != 0: stop aPoller when 'cond' becomes true.
-- removeflag != 0: remove condition from aPoller when 'cond' becomes true
method triggerIf { arg aPoller, cond, anAction, startflag = 0, stopflag = 0, removeflag = 0;
        if stopflag ... (code abridged here, as this is a method with many option combinations ... }
-- start when 'cond.(cval)' becomes true in poller aPoller
-- Options: runfor: duration to run for. startflag, removeflag, stopflag: as in triggerIf.
method startIf { arg aPoller, cond, runfor = 0, startflag = 0, stopflag = 0, removeflag = 0;
        if runfor
        then this.triggerIf(aPoller, cond, {this.runFor(runfor)}, startflag, stopflag, removeflag);
        else this.triggerIf(aPoller, cond, {this.start}, startflag, stopflag, removeflag);
        end.if; }
-- stop when 'cond.(cval)' becomes true in poller aPoller
-- Options: stopflag != 0 stop aPoller, removeflag: remove cond from aPoller
method stopIf { arg aPoller, cond, removeflag = 0, stopflag = 0;
        this.triggerIf(aPoller, cond, {this.stop}, removeflag, stopflag); }
-- Create a new poller and set him watching at a certain object or objects using function 'watchfun' until cond becomes
true. Then perform anAction. Options are like those of 'triggerIf', plus wRate: the polling rate for looking at objects-
tolookat.
method watch { arg watchfun, cond, anAction,
        stopflag = 0, removeflag = 0, wRate = 0.1;
        var checker;
        if cond == 'nil' then cond = { arg cval; ^cval } end.if;
        wRate = wRate max: 0.01;
        checker = Poller(wRate, watchfun);
        this.triggerIf(checker, cond, anAction, 1, stopflag, removeflag);
        checker.start; }
-- watch watchfun with another poller at rate wRate, and start when cond becomes true.
method startWhen { arg watchfun cond, stopflag=0, removeflag=0, wRate=0.1;
        this.watch(watchfun, cond, { this.start }, stopflag, removeflag, wRate); }
-- watch watchfun with another poller at rate wRate, and stop when cond becomes true.
method stopWhen { arg watchfun cond, stopflag=0, removeflag=0, wRate=0.1;
        this.watch(watchfun, cond, { this.stop }, stopflag, removeflag, wRate); }
```

5 Conclusion: Overall Framework, Further Issues, Future Work.

This short paper covers only a few basic aspects of scripting languages for interactive sound applications. Other issues are known and described amongst others in the literature cited, such as the problem of concatenation, scaling and mutual adjustment of events features. Preliminary studies for dealing with these problems have already been conducted within the framework presented here. Another basic issue is that of nested dynamic frames of reference for the scaling of parameters in complex sound structures. One prominent use for this would be for the scripting of pitch and rhythm structures, when the numeric value of a pitch depends on a series of outer reference frames (dynamic transposition / tempo adjustment). A challenge for future work is to build on this basis a framework similar to the VRML scripting language, but for audio.

References

Dannenberg, Roger; Peter Desain and Henkjan Honing. 1997. “Programming language design for music”. In: *Roads, Pope, Piccialli and De Poli 1997*. pp. 271-316.

Pope, Stephen Travis. 1997. Musical Object Representation. In: *Roads, Pope, Piccialli and De Poli 1997* p. 317-347.

Roads, Curtis. 1996. *The Computer Music Tutorial*. Cambridge (Mass.): MIT Press.

Roads, Curtis; Stephen Travis Pope, Aldo Piccialli and Giovanni De Poli (eds.). 1997. *Musical Signal Processing.* Lisse: Swets & Zeitlinger. 1997. pp. 271-316.