

Time and Structure in Xeq

Krzysztof Czaja

Academy of Music, Warsaw

Abstract

Xeq is an external library for PureData, intended for manipulation of sequences of time-stamped Pd messages. It has been used for over two years, both as a flexible qlist replacement and in a number of specific applications (sometimes as exotic, as old-fashioned playing and recording of MIDI files, score following, etc.) In its new version, which is described here, xeq has been entirely redesigned. Although xeq is no longer a qlist extension, nevertheless, it should still naturally match simple tasks, yet prove more suitable for complex musical projects.

Introduction

The novel `xeq` features have been designed to facilitate dealing with time and structure.

There are many different ways of dealing with time, but two are most common in music. Some people tend to use an abstract notion of regular or irregular metrical grid, making tempo changes an integral part of a sequence. Others prefer to work more directly in the domain of real-time flow of events: they measure time with a clock, and, possibly, warp it externally. These two approaches can be combined in `xeq`.

Of all possible ways of dealing with structure, even few most obvious are impossible to contain in a single tool. `Xeq` just picks three that fit easily to the Pd patching and messaging environment. The most important one, is to build a static network of interconnected Pd objects, as a visual representation of parent–children relationships. Another, dynamic, but more lower-level way of manipulating `xeq` structure, is to unfold it into successive sequence-rendering messages (which may be sequenced themselves). Finally, there is even more low-level way — controlling multiple sequence playback objects with another sequence.

Simple mental exercises served as a preliminary testing ground for `xeq`'s design: thinking about patches, that allow to independently control tempo

on two layers. For instance, capability of changing global tempo, while retaining internal characteristics of motives. Or the opposite — making small ornaments more acute, or more smooth, but keeping the flow of narration intact.

On the other extreme, `xeq`'s design has been guided by much more challenging mental exercises: thinking about ambiguous structures, and about ways of maintaining continuity across structural boundaries.

Rudiments

Structural manipulation is not the only reason, why `qlist`-based design of `xeq` has been abandoned.

`Xeq` introduces various random access modes, in addition to forward and backward traversal. This is the most obvious feature lacking in `qlist`, which does not provide any other way of accessing events, than starting from the first event of a sequence, and visiting every next event up to the required one; even accessing the previous event is not possible.

The constraint, that only a single object may perform the traversal, is another major `qlist`'s deficiency. There is only one reading head per `qlist` sequence, while there may be any number of `xeq` readers.

New events may only be appended to the end of a `qlist`'s sequence. `Xeq` events (or entire sequences) may be inserted at a given clock time, or metrical time.

`Xeq` associates a tempo map with every sequence, and maintains time stamps in the metrical, tempo-invariant form, as well as in the form of the clock time, which is warped by tempo changes. Without this duality of time representation, many time-related manipulations could not be easily supported.

Compatibility

The new `xeq` has had to be written from scratch, using internal sequence representation no longer based on the `qlist`'s single buffer of atoms. Moreover, the interface of `xeq` objects has had to be changed, in order to support structural connections. Therefore, old patches are not compatible.

Several old features of `xeq` are still retained in the new version:

- reading and writing of `qlist` files;
- accepting generic Pd messages (with a target) as sequence elements;

- the optional interpretation of Pd messages as MIDI messages, done at a stage of playback, recording, score following, file I/O, etc.;
- using global symbols as sequence identifiers;
- the requirement, that exactly one Pd object (a “host”) owns a particular sequence;
- distributing specific uses of a sequence among several auxiliary objects (“friends”).

Events

A `xeq` event is a Pd message (an array of atoms), or a list of comma-separated messages. It has two additional attributes: a target and a time stamp. The event’s target is a symbol, which identifies receiver(s) of the message. Its time stamp has dual representation: as a clock time measured in milliseconds from the beginning of a sequence, and as a metrical time, measured in bars and ticks.

Some `xeq` objects handle events in a special way, interpreting certain types of Pd messages as MIDI messages. Loading a MIDI file, or recording a stream of MIDI messages, involves the opposite, “MIDI-to-PIDI” conversion.

Specifying time

Many `xeq` messages and objects appear in three variants. For example, the message used for positioning playback objects has the millisecond clock form `goto ms time`, plain bar form `goto bar position`, and metrical form `goto at bar#, beat#, division`. In the plain bar form, an integer part of the *position* argument has the same meaning, as the *bar#* argument in the metrical form — it is the number of a measure. The fractional part is a fraction of that measure. In the metrical form, any two non-negative floats may specify an inside-measure position, where *division* is a number of beats in the whole note. Bar line and beat measurements are zero-based.

Tempo

A tempo map is a sequence of tempo changes. Insertion, deletion, or modification of a tempo map element alters millisecond time stamps of subsequent events, bar lines and tempi, but keeps their metrical time stamps unchanged.

Hosts and friends

For any sequencing task, at least two `xeq` objects have to be created. These are: a server-like object, which “owns” a sequence, and a client-like object, which uses the sequence in any form of playback, analysis, etc. The owner is called a “host”, the client — its “friend”. Both objects refer to the shared sequence by name.

There may be any number of friends using the same sequence, but only one host. Host and friends communicate internally. In particular, whenever a host makes any change to the sequence, it multicasts `pause` and `continue` messages to all its friends. Another example is the `.proxy` friend, which passes all its received messages to the host.

Simple playback

In the simplest case, there are two objects: one loading, and the other performing a qlist file:

```
xeq name
```

```
xeq .qlist name
```

The `.qlist` object (a friend), similarly to the built-in `qlist` original, either traverses the sequence in step mode (triggered by `next` and `prev` messages), or performs the sequence in real-time (after receiving the `start` message). Just like the built-in `qlist`, `xeq .qlist` dispatches messages remotely, while sending millisecond delta times through its first outlet.

The two following objects will play a MIDI file, as soon as the object to the right (a friend) receives the `start` message

```
xeq name
```

```
xeq .pidi name
```

During playback, the first outlet of the friend `.pidi` (parsed MIDI), transmits MIDI note events, in the form of messages accepted by the `noteout` object. The second up to the sixth outlets, pass other MIDI channel messages, the seventh outlet sends a channel number, the eighth sends a track name, and the last one `bangs` when playback reaches the end of sequence. When a `.pidi` object receives a `goto`, or `stop` message, it automatically supplies missing note-offs.

Similarly works the `.rawmidi` friend, except for merging all MIDI events into a stream of MIDI bytes sent through the leftmost outlet (which might feed a `midiout` object).

In both examples, the object to the left is the host, which owns a sequence referred to by *name*. When created, it attempts to load the sequence from a

file *name* (either a MIDI file, or a qlist-compatible text file). The sequence may later be reloaded by sending to the host the message `set filename`, followed by a `bang`. Two more explicit forms of a file-reading host are

<code>xeq name file read filename</code>
<code>xeq name file link filename</code>

Operators

Any `xeq` sequence is a result of some operation. In other words, any `xeq` host is an “operator”, which creates its sequence. Operation arguments may be atoms, or other sequences.

Atomic operation arguments are declared, and initialized, by the operator object’s creation arguments, starting from the fourth argument. Aside from the leftmost inlet, an operator object has as many additional inlets, as there are atomic arguments. The leftmost inlet is used for general control, and for the structural connections.

The values of atomic arguments, but not their types, are modifiable. New values might be assigned, either through additional inlets, or by sending the `set` message to the leftmost inlet. Moreover, some operator types accept specific messages for changing argument values (for example, delta time arguments of the `seq` operator may be `warped`).

An operator object has as many outlets, as there are sequence-arguments it expects. A structure is formed, by connecting these outlets to the leftmost inlets of other operator objects — those that own subsequences. If an outlet is not connected, a corresponding sequence argument is evaluated as an empty subsequence.

Rendering is triggered by an operator object, which receives a mouse click, or a `bang` message sent to its leftmost inlet. Rendering request is then propagated down the structure.

When an operator object receives a rendering request, an operation will be performed — provided, however, that the sequence indeed has to be updated. For most operator types, a sequence is up-to-date (or “fresh”), if both that sequence and all its subsequences have already been updated since last modification of any of their operation arguments. Some operator types may have additional dependencies defined.

The three most basic structural operator types are: `sim`, `suc`, and `seq`. The `sim` operator time-aligns all subsequences simultaneously, by synchronizing their first events. The `suc` operator successively synchronizes each subsequence’s last event to the next subsequence’s first event. The sequen-

tial operator, `seq`, combines subsequences according to a list of delay times. The use of `sim` and `suc` operators may involve marking the start and end points of subsequences with empty events.

Generic operator

An object `xeq name` is a generic operator. Apart from the operation `file read`, which is evaluated at creation time, it accepts messages containing operation requests. Any operation may be requested, using the same syntax, as if supplying specialized operator's remaining creation arguments, following the *name*.

Sequence arguments cannot be provided to a generic operator by means of structural connections. Instead, sequence arguments are explicitly specified as a part of an operation request. Therefore, the structure of a sequence owned by a generic operator is dynamically modifiable.

The `-with` keyword serves as a separator between atomic arguments and a list of subsequence names. If the `-with` keyword is missing, an operation is either performed in-place, whenever possible, or the request is rejected.

It is possible to combine the static and dynamic interfaces to `xeq` operations. A dynamically structured sequence, owned by a generic operator, may both contain and be a part of statically structured sequences, hosted by specialized operators.

Playback control

Another technique consists in using a “master” sequence, or several such sequences, containing playback control events. The targets of those events are `xeq` playback objects. The messages are: `start`, `stop`, `backstart`, `loop`, `backloop`, `goto`, `clip`, `timescale`, etc. A user might choose to build a structure with playback control in search for a more flexible interaction, or if in need for conserving computer memory. However, playback control is a fragile method, and conceptually over-complex.

Recording and saving

Recording, like playing back, is a task for a friend. It is only possible to record into a sequence hosted by a generic operator. Any playback-capable friend of such an operator (like `.qlist`, or `.pidi`) enters recording mode, after receiving the `record` message. A friend, then, initiates a clock, and starts

collecting messages sent to its second inlet. After optionally transforming a message, according to the friend's type-specific rules, it forms an event, by attaching a time stamp and a target. That event is then passed to the host for insertion into the sequence. A target may be specified, either as the friend's third creation argument, or as an argument of the message `target`.

Saving to a file, like loading from a file, is done by a host. All hosts accept the messages `save qlist`, and `save midi` (there is an optional second argument `filename`, which defaults to a sequence name).

Encapsulation and parameterization

The Pd way of dealing with structure is to encapsulate reusable components in abstraction patches. Therefore, facilitating the use of Pd abstractions as user-defined operators, is an important part of `xeq`'s design.

Subsequences contained in a reusable abstraction should either have unique names (e.g. parameterized with `$0`), or be anonymous. Typically, using anonymous operators is more natural, unless an abstraction contains their friends. An anonymous operator is instantiated as `xeq - arguments`.

Efficiency

Structural `xeq` is in an early stage of development. Nothing has been frozen yet. Although several tough design and implementation decisions have already had to be taken, there is still much room for improvement.

The highest priority has been given to the time-efficiency of sequence playback. The cost of accessing the next, or previous event, and of getting the millisecond value of a delay between two subsequent events, is very small and constant.

The cost of recording should also be small. However, it can only be constant up to an arbitrarily specified number of recorded events.

Real-time rendering is the essence of structural `xeq`. This is the whole point — building and transforming structures interactively. The importance of time-efficiency of rendering a structure is only next to that of playback and recording. Therefore, the default behaviour is to keep all, even anonymous, or friendless subsequences in memory, after rendering a structure. Before processing of a rendering request, operator objects check for their “freshness”, and perform rendering only if necessary.

Of course, it is easy to predict, that many real-world structures will fail to render fast enough, so as not to interrupt the flow of audio in heavy sound

processing patches. One option for a user is to explicitly spread rendering process in time, by sending a sequence of rendering messages (several **bang**s to specialized operators and operation requests to generic operators), in order to separately trigger rendering of selected substructures. Another option is tracing and chaining. As the last resort, a user might want to separate audio from sequencing into two Pd processes, possibly running on two different pieces of hardware. Eventually, **xeq** should support threaded rendering, if specifically requested by the user.

Space-efficiency has the very low priority. Currently, the only safety valve is the user, which should either be careful when designing a structure, or otherwise replace parts of a structure with generic operators, requesting them to render subsequences in-place.

Although it is possible to automatically reduce the number of sequences maintained for a structure, any such reduction would cause all operators not maintaining their sequences, to be always “stale”. In future **xeq** versions, users should be able to mark an operator object as “volatile”. An operation of a volatile object would then render either directly into a super-sequence (if certain conditions are met), or into a reusable sequence taken from a pool.

Rendering steps

Rendering is performed, whenever a “stale” operator receives a rendering request. There is also a special case of creation-time rendering, triggered by the constructor of an operator object. Only operators not depending by definition on other operators, perform creation-time rendering (for example, **event**, **note**, or **file**).

Rendering of a structure is triggered by that operator object, which receives a **bang** or a mouse click. Rendering requests are first propagated down the structure, so that before rendering of a sequence, all its subsequences are already rendered.

Prior to the **render** request, triggering operator sends the **discover** message to all its outlets. Upon receiving that message, an operator first passes it down the structure, then determines its own freshness, and finally sends a response back to that object, from which it received the message. There are several reasons for doing so: discovering feedback loops, discovering the names and freshness of substructures, and letting each operator know how many rendering requests it is going to receive, so that rendering into super-structure may be performed.

Internal representation of sequences

Metrical time is represented by a pair: pointer-to-bar, number of ticks after the bar line (a double precision float). Bars may be safely pointed at, because they are never moved around in memory. Bars are readily accessible by index — a standard growable array of bar pointers is maintained.

A dictionary structure (red-black tree) maps clock time to events. Additionally, events are two-way linked. Thus, the dictionary is used for fast insertion, deletion, and searching, while the list is used for fast traversal in both directions. The overhead of synchronizing the two data structures is very small.

A separate dictionary holds the tempo map. The keys are clock time stamps, represented as double precision floats, in the same fashion, as in the case of event dictionary.

The message part of an event may be shared with other events. It is allocated from a reference-counted memory chunk — a buffer of Pd atoms. There may be several chunks referenced simultaneously in a sequence. However, there is always exactly one chunk, the append buffer, from which all sequences allocate atoms for new messages. When the append buffer fills up, a new one is created. The old one is maintained, until its reference count drops down to zero.

Events are rather thick objects. They contain both clock, and metrical time stamps, both tree and list linkage — apart from the actual message. Although speed, precision, and flexibility could not come without a price, requiring on average 1Mb for 10.000 events might turn out too costly for some applications.

Time conversions

Conversion from metrical time to clock time has varying speed. If there is no tempo change after the bar line and before the tick, the conversion is a simple arithmetic calculation, because enough information is maintained for every bar line: both its clock time and its tempo. However, if there are tempo changes in between, the actual tempo at the metrical time has to be found first (by list traversal), then it is a simple calculation again.

The less frequently performed conversion from clock time to metrical time, involves a dictionary search for the tempo valid at the clock time and another search for the bar containing the clock time, followed by an arithmetic calculation.

Conclusions

Pd, and other similar environments, ought to offer more to composers, than just a superficially controllable performance of pre-composed sequences or generative processes. Therefore, the `xex`'s goal is to approach interactive composition as naturally, as possible for a musician, without abandoning the currently most solid framework of interactive computer music. In other words, `xex` should help to explore the middle ground between “orchestra” and “score”.

The tasks for a near future are: establishing a standard set of operators, introducing a looping operator, implementing the most desirable remaining efficiency-related features (the “volatile” property, and, probably, threading), reimplementing and extending the score following utilities. Apart from these, several enhancements are necessary:

- definition of a protocol for the two-way `xex`-gui communication;
- implementation of sequence editors;
- definition of a stable API, and other facilities for specifying additional `xex` operators in a simple C source, in scripting languages, and in lisp;
- introduction of sequence analysis methods.