Viuhka:
A Compositional Environment for Musical Textures

Mikael Laurson
Sibelius Academy, Center for Music and Technology
P.O.Box 86, 00251 Helsinki, Finland
laurson@siba.fi

Abstract

This paper presents a visual compositional tool inside PatchWork (PW, Laurson and Duthen 1989, Laurson 1996a) called Viuhka. This work has been motivated by the need to form bridges between various compositional ideas, visual components found in PW and control of sound synthesis. Viuhka allows to generate complex, multi-layered textures either for instrumental music or for sound synthesis. The user can create Viuhka patches visually by choosing a set of boxes and making connections between them. After feeding appropriate inputs, the user evaluates the patch which results in the final musical texture. The output can be inspected either as a chord sequence or as a piano-roll representation. If necessary the patch can be edited and re-evaluated. As a final step the output can be converted to a synthesis control file.

1. Introduction

Viuhka (Finnish, in English 'Fan', in French 'Éventail') is a PatchWork (PW) user-library designed for and based on ideas by Paavo Heininen (a well known Finnish composer, currently the composition professor at Sibelius Academy). PW, in turn, is a widely used visual programming environment with a strong emphasis on computer assisted composition.

Viuhka is used to create complex, multi-layered musical textures. These can be used to produce scores for instrumental music and to provide material for sound synthesis. The end result of a Viuhka patch can either be displayed in a PW multiseq module or in a PW piano-roll display. Finally, the texture can be translated to a synthesis file (a Csound (Vercoe 1991) score file or a SuperCollider2 (McCartney 1998) patch file).

The starting point is a group of break-point functions (*Viuhka bpfs*) that constitute the pitch field (harmonic skeleton) of the result. Viuhka bpfs provide the main pitch material which can further be reacted to (or elaborated by) different sorts of *Viuhka ornaments*. Besides "typical" ornaments (like grace notes, trills, etc.), the Viuhka ornaments can also be runs, repetitions, chords, clusters, clouds or even complete Viuhka patches (i.e. a Viuhka patch can contain other Viuhka patches). The user typically divides a Viuhka patch into musical time slices or *segments*. The duration of the segments are given as a list of delta-times (or time intervals). The structure of each segment (i.e. number of layers, rhythmic structure, time modification, ornaments and synthesis information) is normally given inside a PW abstraction. There the user defines the *Viuhka parameter values* by using v-params and i-params boxes. The output of the v-params boxes are then given to the main Viuhka box (called mk-viuhka) that in turn calculates the final score.

In the following text we introduce some of the main components of Viuhka. Due to space limitations we will not discuss certain topics - despite of their importance - in great detail. For instance, we will not describe exhaustingly how the user can create Viuhka bpfs for the pitch fields used by Viuhka (there are several tools in PW that allow this: for instance the user can import chord sequences from Finale; chord sequences can be generated with a constraint based approach using PWConstraints (see for instance the fifth chapter in Laurson 1996a or Laurson 1996b), bpfs can be edited by hand, and so on). Also we will not discuss how a Viuhka texture is translated to a synthesis control file as this topic has already been dealt with elsewhere (see Laurson 1999).

This paper attempts to give a rough overview on how the system works mainly from the user point of view. We start by describing in very general terms how to build a relatively complex Viuhka example (sections 2 and 3). After this we go into more detail (section 4) by introducing each Viuhka input parameter in turn. These inputs are used to build segments and layers for the final result. Some of these parameters are quite simple and self explanatory, some of them are, however, very complex - see for instance the "Time-Modif" and "Ornament" subsections - and therefore require a more detailed discussion. In section 5 we give some ideas how the user can design the large overall structure of the result by combining various segments and layers. We end this paper by discussing a complete Viuhka example.

2. Viuhka BPFs

The user can create Viuhka bpfs with a function called make-viuhka-bpfs. make-viuhka-bpfs returns a list of lists of x-values and a list of lists of y-values for bpfs. Figure 1 below gives an example of the make-viuhka-bpfs box inside a patch. The user gives four chords that constitute the input. The resulting bpfs are shown in the lowest multi-bpf box:

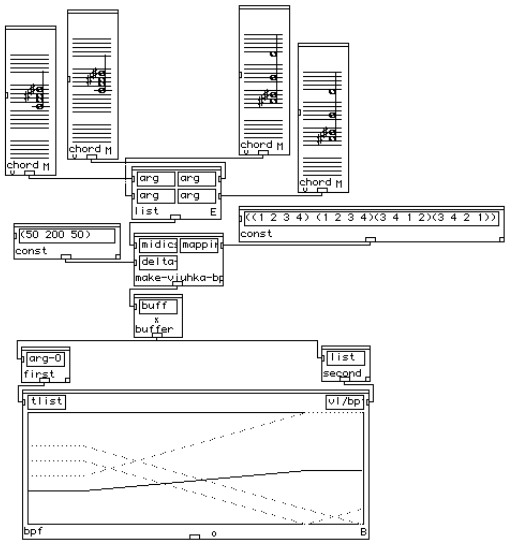


Fig.1: *Creating bpfs with the make-viuhka-bpfs box.*

3. Mk-Viuhka

The primary Viuhka function, called mk-viuhka, returns a list of chord-lines. This output is normally connected to a PW multiseq module and to a synthesis module that in turn creates outgoing synthesis data. The multiseq module allows also to save the result as a midi-file.

The first input, bpfs, is a list of bpf objects defining the pitch field (harmonic skeleton) of the result. The second input, global-dtimes, is a list of delta-times which give the durations of the segments to be produced. The third input, param-ls-ls, is a list of Viuhka parameter value-groups, where each sublist describes the structure of a segment. The patch given in figure 2 shows the main window of a complete Viuhka patch. The patch is split into two parts (the split is indicated by the horizontal dotted line). The upper part shows how the Viuhka bpfs are defined (see also figure 1). The lower part, in turn, shows the mk-viuhka box together with its inputs: the Viuhka bpfs, the list of delta-times for the durations of the segments (global-dtimes) and the Viuhka parameters (params abstraction). Below the mk-viuhka box there are two boxes, multiseq and ->csound, that are connected to the output of mk-viuhka. As can be seen from figure 2, mk-viuhka will use four bpfs (the bpfs input) for the pitch skeleton. The result will have three segments lasting 300, 400 and 250 ticks (1 PW tick = 1/100th of a second) and thus the duration of the whole result is 950 ticks. Finally, the params abstraction defines the structure of each segment.

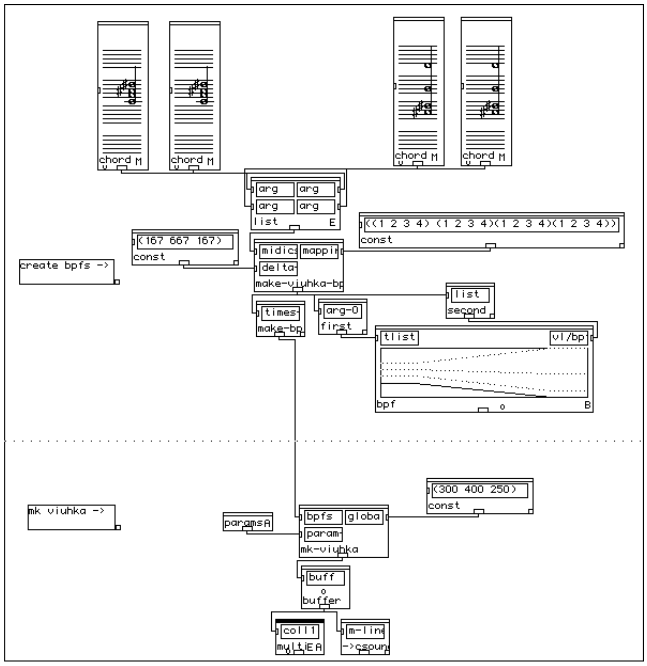


Fig.2: *A complete Viuhka patch.*

Figure 3 below gives the result of the Viuhka patch of figure 2 in a multiseq editor. The example consists of three segments (the segments are indicated by the vertical dotted lines). Each segment has three layers (this feature is defined inside the params abstraction). The example in figure 3 has ornaments such as grace notes, glissando runs and clusters. In order to help to distinguish the ornaments from the *main notes* the main notes are drawn with note stems whereas the ornament notes have no stems. Although figure 3 is quite heavily ornamented the pitch skeleton given by the Viuhka bpfs can been seen clearly (compare the chords and bpfs in figure 2 with the result in figure 3).

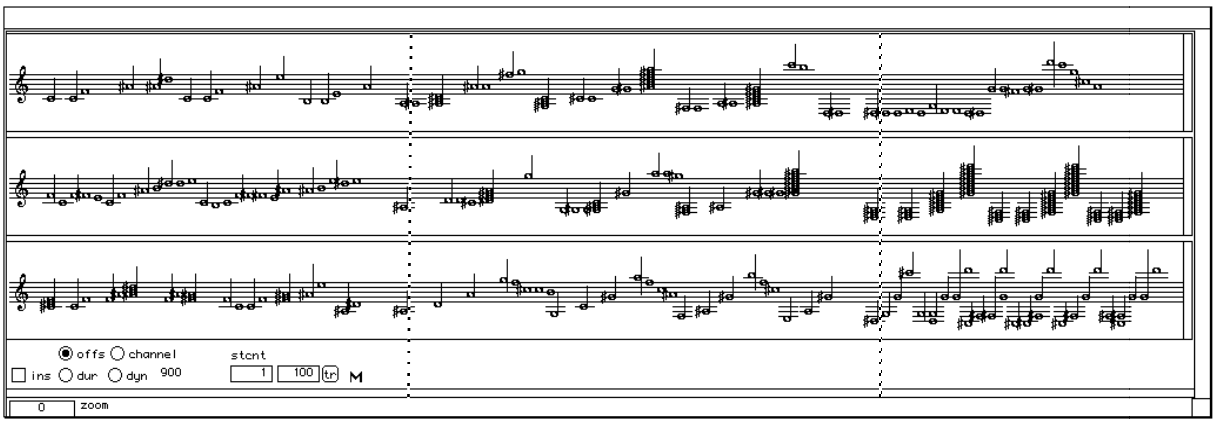


Fig.3: *The result from the Viuhka patch in figure 2.*

4. Viuhka Parameters

In this section we discuss in more detail the Viuhka parameters that are used by the mk-viuhka box to build segments for the final result. A segment can consist of one or several layers. Viuhka parameters for each layer are given in a box called v-params box. A v-params box (figure 4) returns information of bpf indexes, delta-times, time modification, ornaments and instrument data for a synthesis orchestra (in the following we will not discuss the 'instr' input; a discussion of how to interface Viuhka with synthesis environments can be found in Laurson (1999); the last input 'prev-layer' will be dealt with below in section 5).

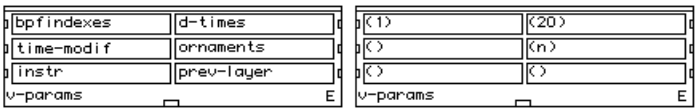


Fig.4: *v-params box.*

Several inputs discussed below accept as arguments circ-lists. Circ-lists are like ordinary Lisp lists except that the items are read in a circular fashion (i.e. after reading the last item, the next item is accessed from the beginning of the list). A circ-list can contain (1) constants (numbers, symbols, etc.) that simply return themselves, (2) *special variables* that are evaluated when they are being read or (3) Lisp expressions. Viuhka contains a collection of special variables that give various sorts of information during the calculation of a Viuhka score. Special variables are typically used inside the input circ-lists and allow the user to specify expressions that react to the current state of the system (such as current time, current pitch, current duration, etc.).

Thus, the circ-list (20) produces a sequence: 20 20 20 20 20 …, and so on; (10 20 30) results in: 10 20 30 10 20 30 10 …; (10 (rand 10 20) 30) results for instance in the following (the second item , '(rand 10 20)', is a Lisp function that returns a random integer value between 10 and 20, inclusive): 10 12 30 10 18 30 10 14 30 10 …; ((if (= mypitch 6000) 'c 'gr) 'n) results in (depending of the value of the special variable 'mypitch'): c n c n gr n …, or gr n gr n gr n …, or c n gr n c n …, etc.

4.1 Bpf Indexes

bpfindexes is a circ-list of indexes that are used to read Viuhka bpfs (given by the first input of the mk-viuhka box).

4.2 Delta-Times

d-times is a circ-list of delta-times (i.e. time intervals indicating when the next event will occur). These values define in conjunction with the time-modif input the rhythmic structure of the segment.

In order to demonstrate the effect of the first two inputs of the v-params box let us investigate two Viuhka examples. Figure 5 below gives a simple Viuhka patch. It consists of only one layer (lasting for 500 ticks and has only one Viuhka bpf - see the ramp in the multi-bpf module at the top of the patch). The params input of the mk-viuhka box is defined by one v-params box. Its first input, bpfindexes, is (1), meaning that we always read the pitch information from the first and only Viuhka bpf. The second input, d-times = (20), controls the delta-times between the notes in the final result (in this case we get a new note every 20 ticks or 5 notes in a second).

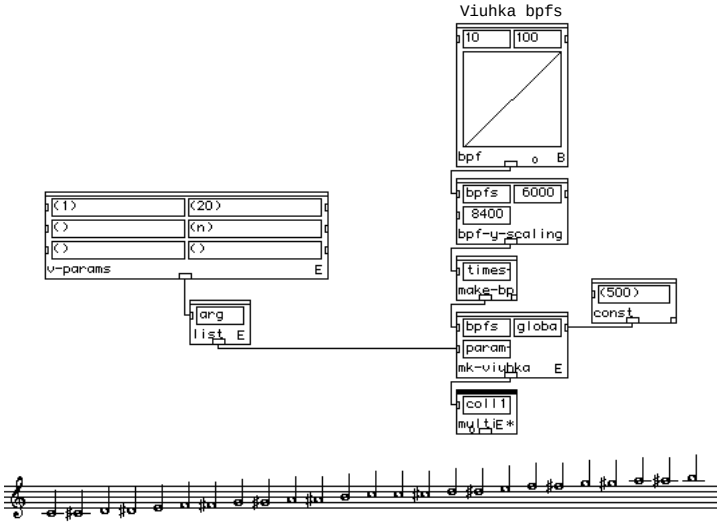


Fig.5: *A simple Viuhka patch with its result.*

Next we modify (figure 6) our example patch by introducing a new bpf to the Viuhka bpfs (see the two ramps in the multi-bpf module). We change the bpfindexes input to (1 2), which means that we alternate between the first and the second Viuhka bpf. Also the d-times input has changed to (20 10 15). This in turn implies that the first note gets the delta-time 20, the second 10, the third 15, the fourth 20, etc.

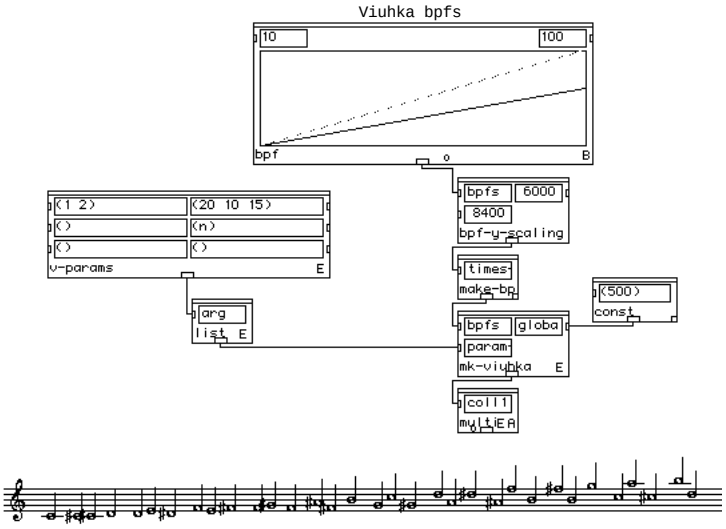


Fig.6: *A modified Viuhka patch with its result.*

4.3 Time-Modif

The time-modif input accepts two kinds of inputs and effectuates two types of time modification within a segment. The two input types are *tempo-bpf* and tempo-bpf combined with a *focus-valu*e. Due to space limitations we concentrate here only on the former input type. Appendix I gives a more detailed discussion of how Viuhka tempo-bpfs are converted internally into *scaler-bpfs* and *time-maps* (Jaffe 1984) and how these are used to modify the delta-times input of a v-params box.

If the time-modif is not given (i.e. it is ()), then the delta-times coming from the d-times input are not modified. If, however, the time-modif input is given, it must be a bpf object (i.e. a tempo-bpf). The bpf acts as a tempo-bpf modifying the behaviour of delta-times. The y-values of the tempo-bpf are considered to be metronome values (for instance, to double the speed one must double the current metronome value). The tempo-bpf has some special properties as it allows to control the exact timings where a given tempo change will occur in the final result. In a conventional score the performer has usually to realise the indicated accelerandi or ritardandi by playing *all* notes within the tempo change. Also one must gradually reach the desired end tempo when arriving at the last note within that group. The performer thus controls the tempo change (both the start tempo and end tempo) and the number of notes (normally the performer has to play all notes). This means that there is no direct control *when* the end point of the tempo change is reached. It is in a sense a “by-product” of performing an accelerando or a ritardando.

As in the conventional score, the tempo-bpf in Viuhka controls both the start and end tempo when realising a tempo change. There is, however, no direct control of the number of notes that will appear within the tempo change. This in turn allows the tempo-bpf to control the exact time points for tempo changes. Also the user controls directly the duration of the resulting segment.

To clarify this let us investigate a variation of the patch given in figure 6 above (see figure 7). The second ramp of figure 6 is modified so that it is merged with the first one for the first 250 ticks. After this it begins to descend rapidly and produces a gap between the ramps. The third input (time-modif) of the v-params box is connected to a multi-bpf module. The tempo-bpf makes an accelerando (from 60 to 120) for the first 250 ticks, after this it makes a ritardando (from 120 to 60). The metronome values are given to the tempo-bpf by the const-

box. The d-times input has been changed to (20) in order to clarify the tempo change of the result. The interesting point to note here is that the timing of the “split-point” of the Viuhka bpfs and the tempo change from accelerando to ritardando are automatically synchronised. Furthermore, the duration of the resulting segment is exactly 500 ticks:

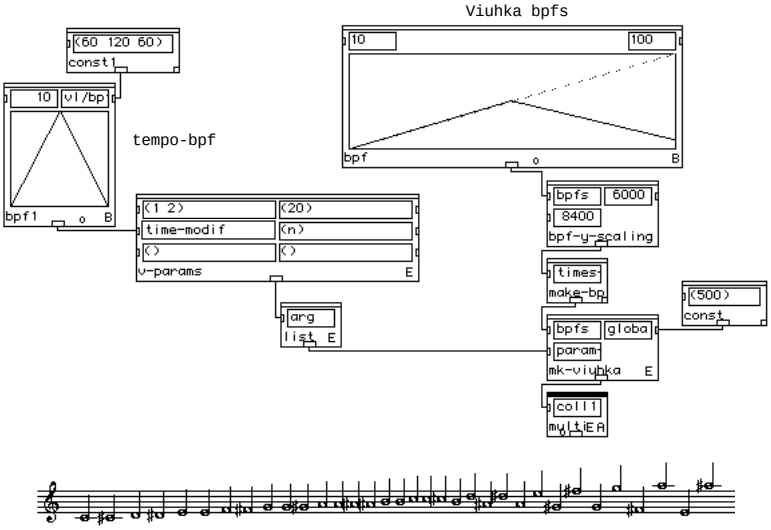


Fig.7: *A tempo-bpf Viuhka patch with its result.*

4.4 Ornaments

The ornaments input is a circ-list defining the ornaments to be used when building the score. There are two basic ways of indicating the type of ornaments. The first way is to use short symbols (like ‘n’, ‘g’, ‘rn’, ‘c’, etc.) that refer to relatively simple ornaments in a predefined *ornament library*. The other way is to give a pointer to a multiseq module. This module in turn evaluates its input (typically a patch) and returns its contents. This scheme is very powerful as it allows to evaluate dynamically any patch of arbitrary complexity. These kind of nested patches will be referred henceforth as *super-notes*. Often the patch to be triggered is actually also a Viuhka patch which means that any Viuhka patch can contain other Viuhka patches. Figure 10 gives a Viuhka patch containing some ornaments from the current ornaments library. The upper staff shows the result with ornaments. The lower staff (containing only the main notes) is given below the upper one in order to clarify how the ornaments where calculated by the system.

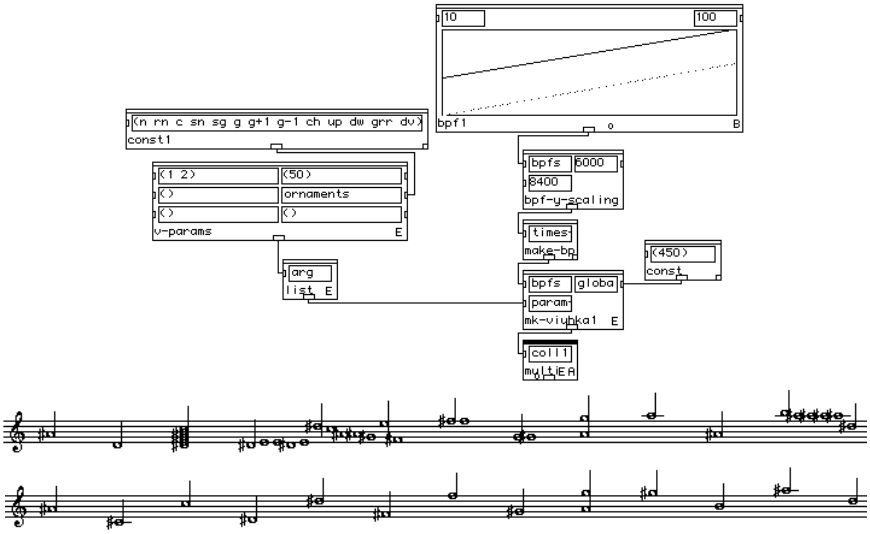


Fig.10: *A Viuhka patch with ornaments from the ornaments library.*

Super-notes (i.e. ornaments that are defined by a patch) are interfaced with the Viuhka main patch by a box called box-ptr (= “box pointer” box). It accepts one input which is in our case always a multiseq module. It returns as a value the CLOS pointer to the multiseq editor. The pointer is used in turn to retrieve the chord-line information that has been produced by the super-note patch. This means that the circ-list, defining the ornaments input of the v-params box, accepts besides symbols (indicating an ornament from the ornament library), also pointers to multiseq modules. Typically a super-note patch needs to access information of the global state of the main note (like ‘mypitch’, ‘mydur’, etc.). Therefore a super-note patch often contains evconst boxes (PW boxes that evaluate their inputs) that in turn contain names of Viuhka special variables.

To clarify this let use investigate a super-note example. As our example will be more complex than the previous ones we split it into two parts: the main patch (figure 11) and the abstraction (figure 12) containing the v-params box and the super-note definition. The main patch is very similar to the ones in the previous examples, except that the params input is defined within an abstraction (see the params abstraction in figure 11):

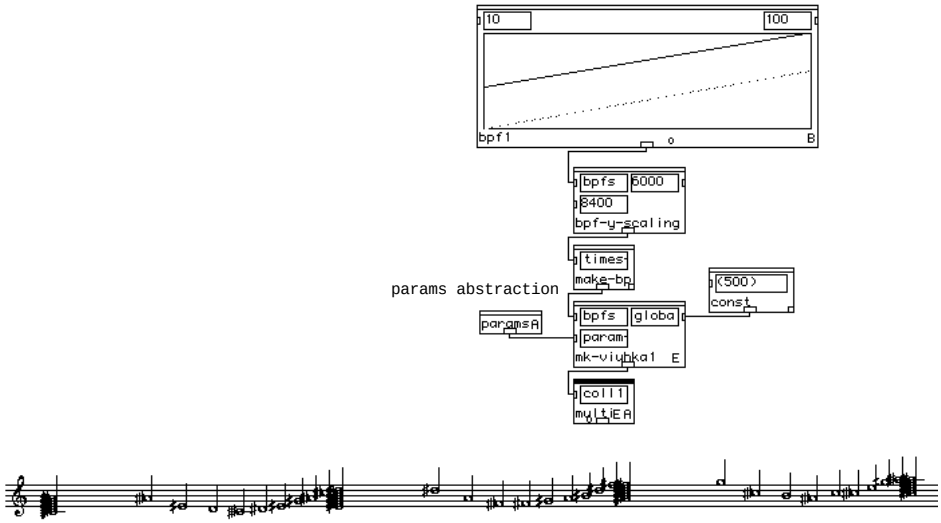


Fig.11: *A super-note main patch with its result.*

Figure 12 below gives the contents of the params abstraction of figure 11. Figure 12 is split into two sections (see the horizontal dotted line). The lower part contains the v-params box of the main patch. Its ornament input is connected to a list-box. Its first input, in turn, contains the symbol ‘c’. The second input is connected to the box-ptr box of the upper part of figure 12. This means that the resulting Viuhka score (see figure 11) will alternate between a cluster (given by ‘c’) and a super-note ornament defined in the upper-part of figure 12.

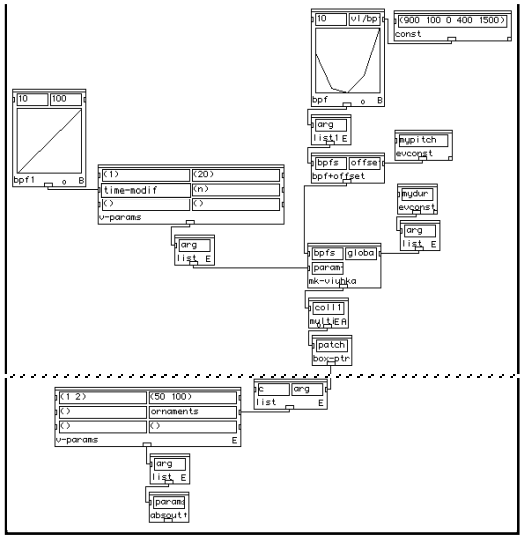


Fig.12: *The super-note abstraction.*

5. Viuhka Segments and Layers

In this section we present how the user can build the overall formal structure of a Viuhka result by chaining v-params boxes - the v-param box was explained in the previous section - both vertically and horizontally. Typically a Viuhka result consists of one or several segments. Segments, in turn, can consist of one or several layers where each layer is defined by a v-params box.

A segment is built by chaining v-param boxes vertically. This is done by connecting in cascade the 'prev-layer' inputs of several v-param boxes. Finally, all segments are collected horizontally to a list-box where the first input defines the first segment, the second input the second segment, and so on. Figure 13 contains a patch that has two segments (the list-box has two inputs). Each segment, in turn, contains three layers:

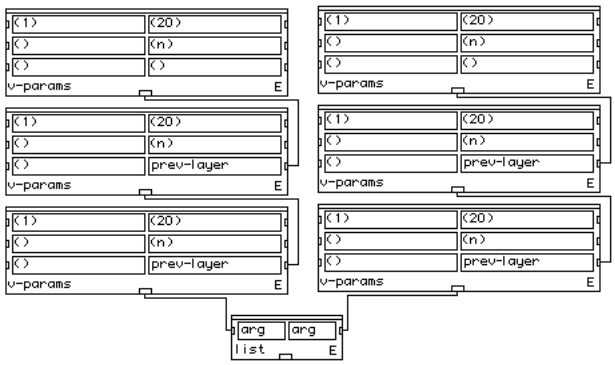


Fig.13: *A patch with two segments and three layers.*

Thus, the patch defining the segment and layer structure of a Viuhka patch forms a kind of a matrix (or *segment/layer matrix*). All segments must have an equal number of layers. This does, however, not mean that all layers within a segment must be "active" during the calculation. A layer that does not participate in the final result will be defined by using a special box called empty-layer box.

6. Viuhka Example

To conclude this paper we will shortly present a complete - albeit relatively short - Viuhka example called "Vocalise5" realised and composed by Paavo Heininen. It is 90 seconds long and is divided into 16 sections. Each section consists of 9 layers.

Appendix IIa gives the Viuhka bpfs used for the example. Appendix IIb, in turn, shows the resulting texture (we do not use here the chord-sequence representation as the piano-roll display shows more clearly the overall formal structure of "Vocalise5"). As can be seen fairly clearly from Appendix IIb, the example contains actually only 5 different texture types which are repeated several times during the piece. In the realisation the texture types were built as PW abstractions only once and were then "reused" several times while calculating the final result. It is interesting to note how these texture types react differently when they are situated at different time positions within the global pitch field formed by the Viuhka bpfs (compare Appendices IIa and IIb).

Conclusions

We have discussed the main components and features of a PW user-library called Viuhka. The focus has been in showing how the system works from a user point of view. We hope that this paper demonstrates that such a compositional system - although it has been originally designed for an individual composer - can have general interest and significance. Viuhka provides some interesting and novel features (see for instance the discussions in the "Time-Modif" and "Ornaments" subsections). Furthermore, Viuhka is unique in PW as it provides a fairly complete system allowing the user to work both with local musical aspects and global musical entities within a single environment.

Acknowledgements

This work has been supported by the Academy of Finland in project "Sounding Score - Modelling of Musical Instruments, Virtual Musical Instruments and their Control". The author wishes also to thank Prof. Paavo Heininen (Sibelius Academy) for his help in formalising and developing the compositional ideas behind the Viuhka project.

References:

Jaffe, D. Ensemble Timing in Computer Music. *Computer Music J.* , Vol. 9, No. 4, pp. 38-48, 1985.

Laurson, M. and Duthen, J. PatchWork, a graphical language in PreForm. In *Proc. ICMC'89*, pp. 172-175, 1989.

Laurson, M. PATCHWORK: A Visual Programming Language and Some Musical Applications; Doctoral dissertation, Sibelius Academy, Helsinki, Finland, 1996a.

Laurson, M. PWConstraints Reference Manual; IRCAM, Paris, France, 1996b.

Laurson, M. PWCollider — A Visual Composition Tool for Software Synthesis. In Proc. *ICMC'99*, pp. 20-23, Beijing, China, Oct. 1999.

McCartney, J. Continued Evolution of the SuperCollider Real Time Environment. In *Proc. ICMC'98*, pp. 133-136, 1998.

Vercoe, B. Csound. A Manual for the Audio Processing System with Tutorials. Media Lab, M.I.T., USA, 1991.

Appendix I: Time modifications in Viuhka.

In this appendix we discuss in more detail how Viuhka time modifications - allowing the direct control of the exact time points for tempo changes - are accomplished. Figure 14 shows how a tempo-bpf (the bpf to the left) is first translated internally into a scaler-bpf (the bpf to the right).

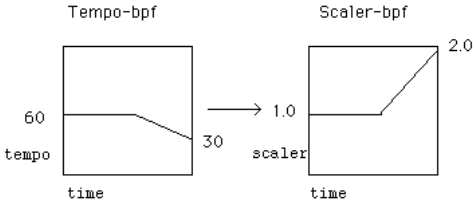


Fig.14: *A tempo-bpf defined by the user is translated into scaler-bpf.*

After this, the scaler-bpf is sampled with a dense tick stream (the default sampling rate is 0.01 s) and the resulting values are accumulated into a time-map (Jaffe 1984). Thus, the resulting time-map is an integral of the scaler-bpf (figure 15). While sampling the scaler-bpf, the system checks whether a note falls within the current time slice or not. If this occurs, a new time value for the respective note is found by reading the y-value of the time-map at the current time point. Thus in our example the time values for the notes at the beginning part are not changed, because the tempo is kept static (tempo = 60 - see the diagonal line segment part of the time-map). The time values belonging to end part, however, are getting gradually larger as the tempo slows down from 60 to 30 (the curved part of the time-map).

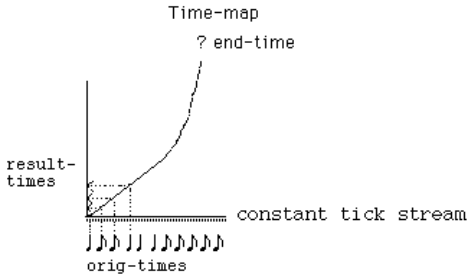


Fig.15: *The accumulated values gained from the scaler-bpf results in a time-map.*

Normally, while sampling the scaler-bpf, the tick stream sampling rate is kept constant (say 0.01 s). Viuhka, however, has the option where the sampled scaler value affects directly the sample rate of the system (figure 16). This is done by multiplying the current scaler value by the sampling rate. In our example, at the beginning the sampling rate (0.01) is multiplied by 1.0 (sr = 0.01), whereas at the end it is multiplied by 2.0 (sr = 0.02). Thus, this means that while the tempo is getting slower the sampling rate is increased causing the system the reach the end time of the segment sooner than using a constant sample rate. The reverse is true if we accelerate the tempo (that is the system will sample slower due to a smaller sample rate). The net effect is that Viuhka allows to control besides the tempo change also exact time positions of the tempo change (the number of notes, however, cannot be controlled directly).

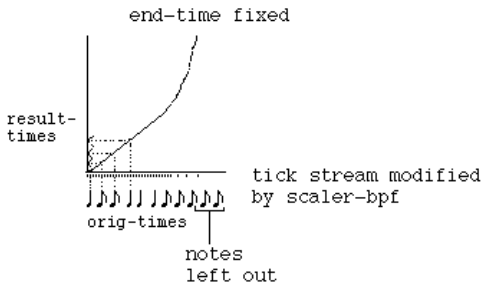
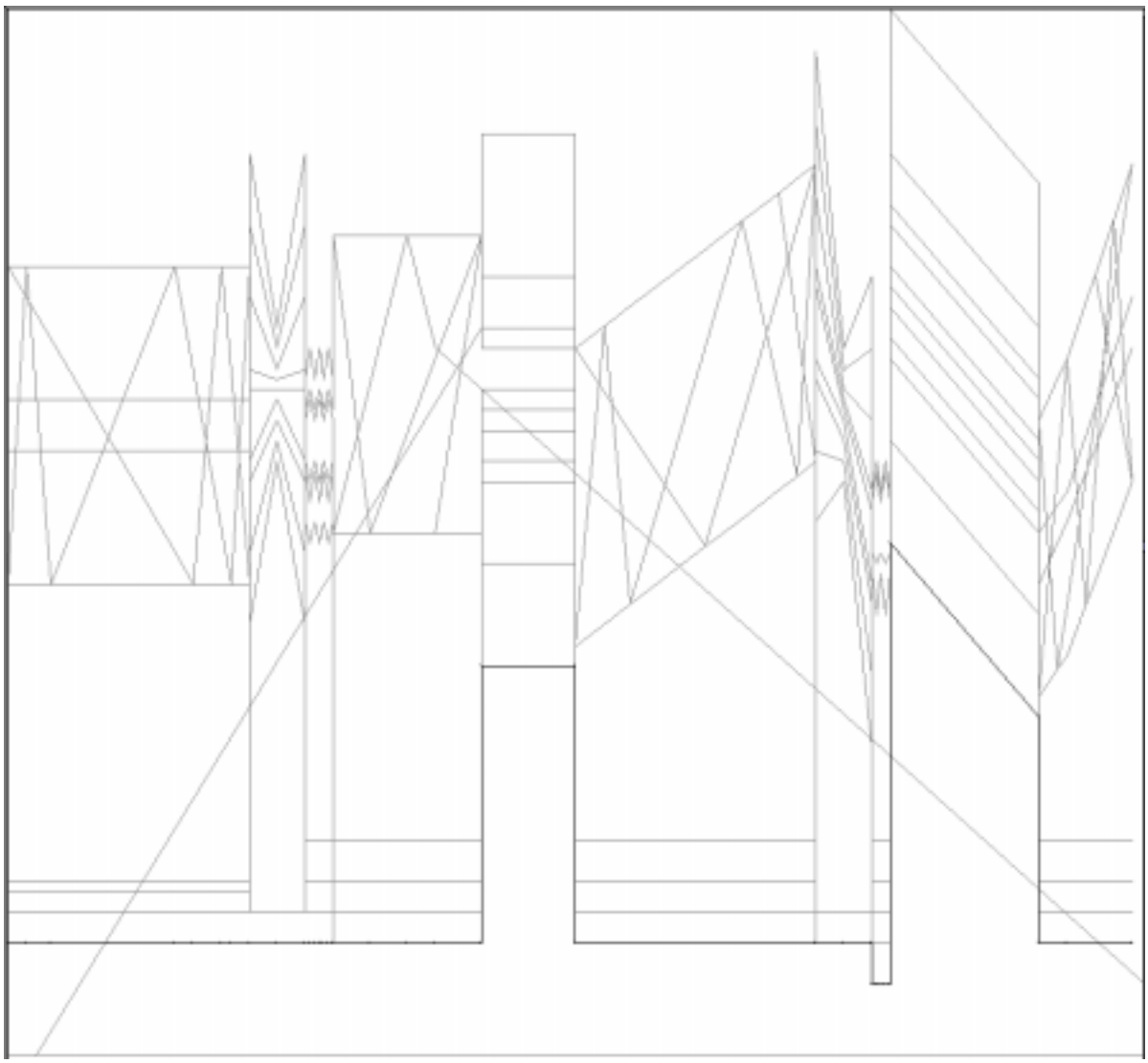


Fig.16: *A Viuhka time-map with a variable tick stream sample rate.*

Appendix IIa: Viuhka bpfs used for “Vocalise 5”.



Appendix IIb: Resulting Viuhka texture as a piano-roll representation.