Overview of the Surmulot Music Composition System

Stéphane Rollandin
hepta@zogotounga.net

draft - 30 March 2012

Abstract

We present the Surmulot system for computer-assisted music composition. This is a dual system with two main entry points, the Emacs[1] editor and a Squeak[2] image, a Smalltalk IDE[3]. Emacs allows for interfacing with additional software, such as Csound[4], while Squeak implements an extensive collection of objects called µO, for "musical objects". These include representations for elementary musical elements sharing a mixing algebra, complex models of musical concepts build upon them, and interactive editors of many kinds making full use of the powerful morphic[5] GUI[6] paradigm unique to Squeak.

1. Purpose

Surmulot is a personal research project ; as such, it does not attempt to meet commercial quality standards in terms of documentation, design or interface. It has been in active development since 2003 and is very rapidly changing in its scope and features.

Nevertheless it is today production-ready in its own sense : it works, is stable, and can be used at many different levels, from rapid prototyping of musical ideas to extensive programmation enabling the user to tailor its own musical tools or implement its own composition paradigm.

2. Architecture

The two Surmulot main components are dynamic IDEs linked via local TCP/IP[7]. One of them is the well-known Emacs editor. Emacs controls subprocesses such as the Csound soft synthesizer or utilities like sox[8], abc2midi[9], wavesurfer[10] and other ancilliary programs. The other IDE is Squeak, an image-based Smalltalk implementation. Squeak is the place were musical composition concepts are reified and rendered interactive in many unique ways. When the Squeak image is saved, all its contents is preserved automatically : it behaves as a permanent workshop where the composer can store and play with all of its musical data without having to worry about having to manage it for practical look-up and retrieval ; everything is always there and can be backed-up in one shot simply by renaming the Squeak image.

The two IDEs are linked so that it is possible to evaluate Emacs Lisp expressions in Squeak, while it is also possible to evaluate Smalltalk expression in Emacs. This way we can harness at best the complementarity nature of their world-views, text in Emacs and live objects in Squeak : graphics can be dynamically generated by Squeak and inserted in Emacs buffers, MIDI data can be exchanged seamlessly, Emacs subprocesses can be launched from Squeak, specialized Squeak image clones can operate as graphical editors for relevant Emacs buffers contents while reciprocally specialized Emacs buffers can edit text-based data (such as ABC file or Csound scores) in behalf of Squeak in which the data is actually maintained. All of this is automatized so that the composer gets the feeling of an integrated, single system.

3. Documentation

Because of its highly dynamic nature and fast evolution, documentation is composed of either general descriptive papers such as the one you are now reading or collections of pointers to very specific pieces of documentation living directly in the system : Info nodes or documentation strings in Emacs, interactive book morphs (some of them analog to Mathematica[11] or Sage[12] notebooks) in µO. The welcome buffer in Emacs acts as the main entry point for documentation and provides all important pointers.

4. Availability

Surmulot is composed of open-source software. It is freely available from the Internet. It runs in Windows and Linux systems while at the moment only part of it are available for MacOS, were it will not behave as an integrated system. Current status is detailed on the

1 http://www.gnu.org/software/emacs/
2 http://squeak.org/
3 Integrated Development Environment
4 http://www.csounds.com/
5 http://en.wikipedia.org/wiki/Morphic_(software)
6 Graphical User Interface
7 Transmission Control Protocol
8 http://sox.sourceforge.net/
9 http://abc.sourceforge.net/abcMIDI/
10 http://sourceforge.net/projects/wavesurfer/
11 http://www.wolfram.com/mathematica/
12 http://www.sagemath.org/

website[13].

4. Overview of µO

µO stands for "musical objects" : it is the name of the collection of classes implemented in Squeak for music composition.

4.1. The mixing algebra

The fundamental paradigm for musical representation used here is mixing : any musical element is a subclass of `MusicalElement` and as such obeys a mixing algebra. A musical element is always anchored in a local temporal frame extending from time 0 to an arbitrary duration. Relatively to that frame, the musical element always occupy a definite position, starting at a given arbitrary time and ending at another arbitrary time.

The elementary operator `delay`: moves the musical element in its referential frame. The operator `scale:` stretch it with respect to that frame. The operator `duration`: sets the length of the frame[14].

Two musical elements can always be mixed : their temporal frames are merged. This fundamental operation either results in a single musical element similar to the initial ones, such as the merging of `MusicalNote`s into a `MusicalPhrase`, or, when the mixed elements are not of the same nature, in a `CompositeMix`, where the initial musical elements are simply referenced at their respective positions. Whenever we want to keep explicit the internal structure of a the result of a mixing operation it is always possible to ask for a `CompositeMix`. Reciprocally, we can always send the `compute` message to a `CompositeMix` so that it gets converted into a simpler musical element, when possible. Other, more sophisticated convertions can be performed so that a `CompositeMix` gets interpreted as a another musical element, such as a `SoundElement`.

The following diagrams will illustrate the fundamental operators. We will represent musical element A by the diagram

```
[  music A ]
------------
0          dA
```

and musical element B by the diagram

```
[    music B    ]
----------------
0              dB
```

13 http://www.zogotounga.net/surmulot/surmulot.html
14 This framework is inpired by the representation of musical phrases in KeyKit, by Tim Thompson : http://nosuch.com/keykit/

Both A and B are simple in the sense that they fully occupy their own temporal frame: their start time is 0 and their end time is equal to their frame duration.

The effect of `delay:`, depending on weither its argument is positive or negative, can be

```
            [  music A ]
-----------------
0              dA+delay
```

or

```
   [  music A ]
---------
0      dA-delay
```

that is an offset of the musical element within its own temporal frame, along with the modification of that frame duration.

The combination of delay and mixing allows the straightforward definition of three operators :

1) The concatenation operator `+` (also expressed as `,`).

The diagram for A+B (or A,B) is

```
[  music A ][    music B    ]
----------------------------
0          dA                  dA+dB
```

while B+A (or B,A) looks like

```
[    music B    ][  music A ]
----------------------------
0               dB         dA+dB
```

2) The mixing operator `|`

A | B is equal to B|A and looks like

```
[  music A ]
[    music B    ]
----------------
0          dA   dB
```

3) The backward mixing operator `//`

A // B is

```
     [  music A ]
[    music B    ]
    ----------------
dA-dB 0              dA
```

while B // A is:

```
    [  music A ]
[    music B    ]
----------------
0              dB
```

Thus backward mixing is actually mixing after having aligned the second element duration with the first element duration.

Now let's consider two other musical elements those structure is not as simple as A and B : the diagrams for musical element C and D are

```
   [  music C ]
---------------------------
0                         dC
```

and

```
     [  music D ]
-----------------
0               dD
```

Since concatenation is relative to temporal frames the diagram for C,D is

```
   [  music C ]                  [  music D ]
--------------------------------------------
0                         dC           dC+dD
```

while D,C looks like

```
     [  music D ]   [  music C ]
--------------------------------------------
0               dD                     dD+dC
```

C | D is again the same as D | C (mixing):

```
   [  music C ]
     [  music D ]
---------------------------
0               dD        dC
```

C // D (backward mixing) results in

```
   [  music C ]
               [  music D ]
---------------------------
0                         dC
```

while the diagram for D // C is

```
     [  music C ]
                 [  music D ]
  ---------------------------
dD-dC       0               dD
```

Scaling a musical element happens relatively to its temporal frame, as illustrated by the following diagrams : the element

```
    [  music ]
----------------
0            duration
```

scaled to twice its duration becomes

```
        [      music       ]
--------------------------------
0                          duration
```

When we add the possibility of scaling, two more basic operators follow naturally: the synchronising operator `&` where A&B is A mixed with B scaled to A duration,

```
[  music A ]
[  music B ]
------------
0          dA
```

and the beat concatenation operator `,,` where A,,B is A followed by B scaled to A duration.

```
[  music A ][  music B ]
------------------------
0          dA         2*dA
```

C&D looks like

```
   [  music C ]
        [      music D    ]
---------------------------
0                         dC
```

and D,,C looks like

```
     [  music D ] [  music C ]
----------------------------------
0               dD                  2*dD
```

Many other ways to modify the structure of a musical element exist which we will not review here.

4.1. The main `MusicalElements`

`AtomicElements` subclasses gather everything with the nature of a single musical note. This includes the usual note as we know it (to keep it short), drum strokes, MIDI events, elementary OSC messages, single Csound score statements, single bols[15] and more exotic objects like projections (see below).

`MusicalCollections` subclasses are collections of `AtomicElements` : a `MusicalPhrase` is a collection of `MusicalNotes`, a `CsoundScore` gathers `CsoundNotes`, a `BolPhrase` is a list of `BolWords`, etc.

By far the most important musical element is the musical phrase. A lot of work has been put in its string representation, so that it is actually by itself a very compact domain-specific language of sort. This is not the place to describe it[16] ; let's just say that it is an hypertrophied version of the KeyKit format for musical phrases.

A major chord would be expressed as `'c,e,g'` or `'c:maj'`. The application of the `fractal` operator on

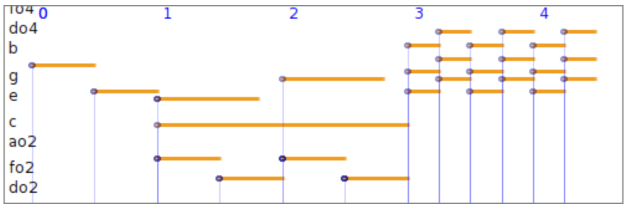
15 Mnemonic syllables originally used in Indian music
16 See "The String Representation of Musical Phrases in µO" paper

the chord would be `'/fractal/ c:maj'` and be equivalent to `'cd32,e,g,e,a-,b,g,b,do4'`.

A more complex and less readable phrase like

```
'a,f,
{c"&&}{go2^,e,g^,e}{/nonleg/ e^&,g""},
{/r3/ f!":min,g}'
```

would look like this in a phrase editor:



Among the important musical elements are envelopes. In µO envelopes are first-class objects not bound to a specific usage. They can be used to shape melodies or pitch inflexions, modulate amplitudes, define accelerando or spectral variations of a chord, bound stochastic distributions, be converted into csound tables, etc.

Since they obey the musical element algebra, they can be concatenated and mixed ; they can also be programmatically defined, interactively edited and altered in many different ways.

Envelopes are made of breakpoints and interpolation functions. Because functions are themselves first-class objects in µO, any form of interpolation can be defined ; moreover, functions can be converted in envelopes, and envelopes become functions[17].

`RhythmicCells` are another important class of objects. They are the building blocks of `RhythmicCanvases` , upon which music can be structured in time.

For example, the simple 4/4 time signature is implemented as a rhythmic canvas of one single cell defined by the Smalltalk code `#((2 2) 4) sig`.

Again, because rhythmic cells obey the musical element algebra they can be combined arbitrarily, making it very easy to create complex time signature and elaborate canvases, then possibly shape them interactively or programmatically . This is described in another paper[18].

17 See for example in the Csound Journal : http://www.csounds.com/journal/issue15/shaping_envelopes.html
18 See "The Representation of Rhythmic Structures in µO"

4.2. Modes and scales

In µO a scale (instance of class `Scale`) is an object maintaining a list of intervals. There is no notion of absolute pitch in a scale (no anchoring in the frequency domain). A scale can span less or more than an octave. Note that a µO scale is identical to what is called scale in the Scala software[19]; the Scala format is supported.

A mode (instance of class `Mode`) is another object that builds upon a scale by selecting only part of it: it maintains a list of steps. Besides, a mode also maintains a root frequency, which is the frequency associated with the zero index in the list of steps, also called the tonic. The tonic anchors the mode underlying scale in the frequency domain.

This is described in detail elsewhere[20]. We will only give a few examples here to give a taste of the way we can work with modes.

The Smalltalk expression

```
(Mode major readPhrase: 'c,e,g') pitches
```

returns the list

```
#(261.6255653005987
  329.62755691287
  391.9954359817494)
```

In the following statement we change the tuning and the temperament of the mode by setting its associated scale to Serge Cordier's perfect-fifths temperament[21] and by specifying an historical tuning convention[22] :

```
((Mode major
   scale: ChromaticScale perfectFifths)
     withHandelFork readPhrase: 'c,e,g')
pitches
```

now returns

```
#(250.8555226443194
  316.2621669801326
  376.283283966479)
```

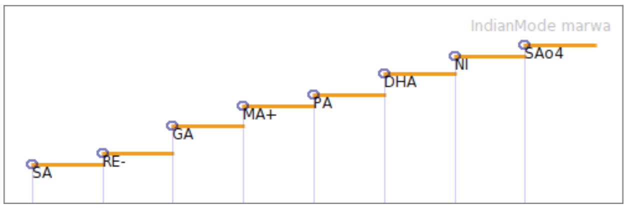
Diatonic modes of the chromatic scale implement a wealth of methods. A few examples :

`Mode G dorian V7` is proper Smalltalk code and return the dominant seventh chord `'do4 f+ a co5'`.

`Mode C major asMode: #lydian` is also proper code and return `Mode F lydian`.

19 by Manuel Op de Coul: http://www.huygens-fokker.org/scala/
20 See "Modes and Scales in µO"
21 S. Cordier: "Piano bien tempéré et justesse orchestrale", ed. Buchet-Chastel 1982
22 See http://www.piano-tuners.org/history/pitch.html

Common Indian modes are also implemented, using a sargam[23] naming scheme for notes. For example, the code `IndianMode marwa view` returns the following figure:



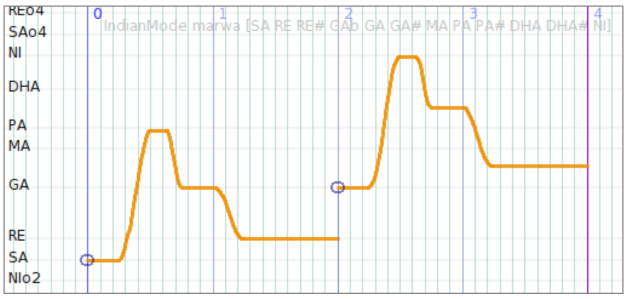
4.3. Inflected notes

Musical notes can have a varying pitch. It is possible to convert a musical phrase into a single note, with a precise control of the shape of the resulting inflexion.

Here we define in turn a mode with a non equal-tempered scale, a phrase built on that mode, then an inflected note built on that phrase :

```
mode:= IndianMode marwa sansAccidentals.
ph := mode readPhrase: 'SA, PA!, GA&, RE.'.
note:= ph staccato asInflectedNote: #smooth.
```

When such a note is transposed the inflexion shape is modified in order to reflect the actual intervals it now covers ; it is kept in tune. Below is `note` and its transposition two mode steps up in a phrase editor. See how the inflexion is aligned to the grid (where it does not seem aligned, it is actually aligned to the underlying scale steps) :


REo4
SAo4
NI
DHA
PA
MA
GA
RE
SA
NIo2
0
1
2
3
4
IndianMode marwa [SA RE RE# GAb GA GA# MA PA PA# DHA DHA# NI]

The amplitude of a note can also be modulated by an envelope, allowing for example the removal of the note attack.

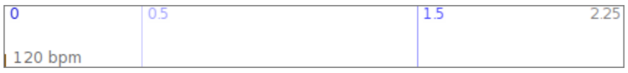
4.3. Rhythm

Rhythm can be considered locally or globally : locally, the setting of a musical element duration defines which span of time the element claims for itself, so concatenating elements of different durations creates a local rhythm ; more globally, two appoaches are implemented : the first one is a generalization of the concept of time signature called the rhythmic canvas, and the second is a representation of time in two dimensions where rhythm unfold from a geometric structure via projections (this is the continuation of the work started with GeoMaestro[24], see below).

Rhythmic cells can bridge the two global models and the local notion of rhythm : we can get a cell from a musical phrase (basically it amounts to extract note onsets from the phrase, although this may be refined), and we can assemble cells into a rhythmic canvas.

Here is how a cell can be created :

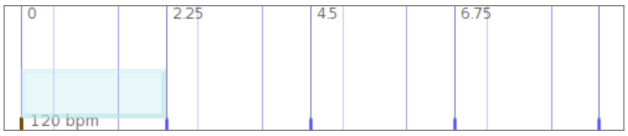
```
cell := 'c,e&,g!.' kphrase asRhythmicCell.
(cell % 2) weak
cell markDownBeat.
```



The cell is illustrated above. Time 0 is the down beat, the strongest possible time. Time 0.5 is a weak time, time 1.5 is strong.

We get a rhythmic canvas from the cell with

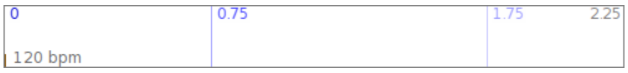
```
canvas := cell asCanvas.
```



`canvas` is illustrated above ; it structures time by duplicating `cell` everywhere : in that sense `cell` is a time signature.

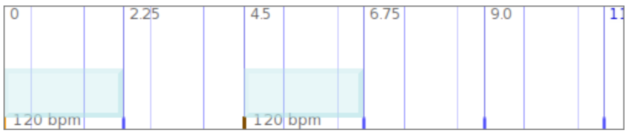
Let's create another cell :

```
cell2:= cell cc reverse.
```



We can add it to the canvas :

```
canvas addCell: (cell2 delay: cell length*2)
```



`canvas` now features a change of time signature at time 4.5.

For more about rhythmic canvases and how to play with them, please refer to the paper "The Representation of

23 http://en.wikipedia.org/wiki/Swara

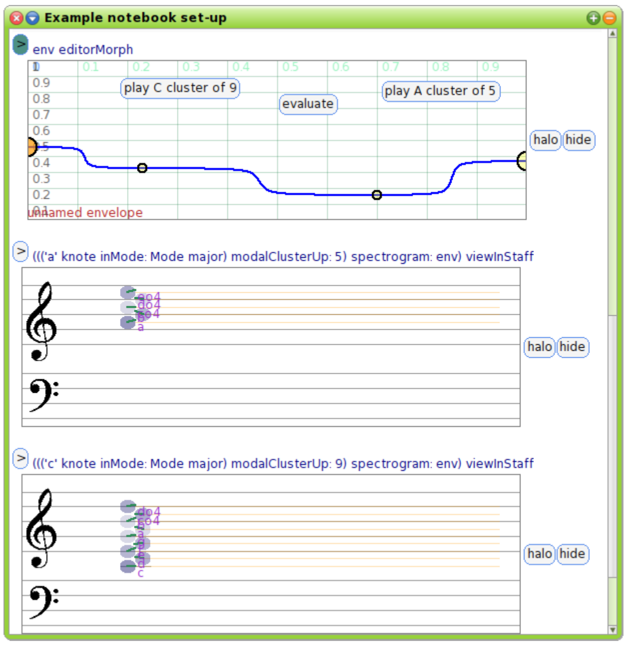
24 http://www.zogotounga.net/GM/eGM0.html

Rhythmic Structures in µO"

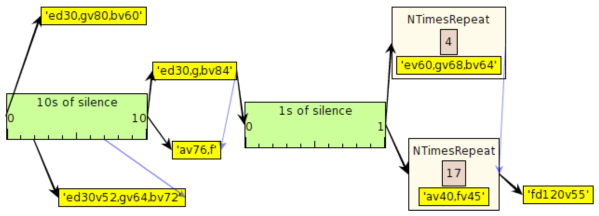
4.4. The main µO interfaces

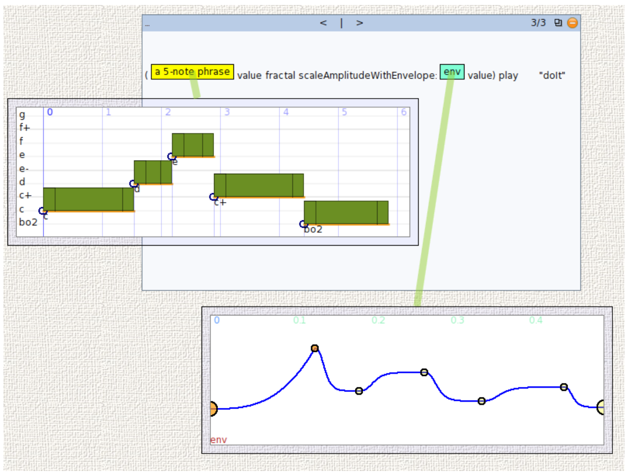
µO is an extension to Squeak ; the actual user interface is the Squeak IDE, that is the Squeak image itself. This is a very unique environment worthy of a lengthy discussion which we will not have here[25].

Among the tools specific to µO



```
((('a' knote inMode: Mode major) modalClusterUp: 5) spectrogram: env) viewInStaff
((('c' knote inMode: Mode major) modalClusterUp: 9) spectrogram: env) viewInStaff
```





```
( a 5-note phrase  value fractal scaleAmplitudeWithEnvelope: env  value) play    "doIt"
```

4.5. The main editors

4.6. Synthesizers

4.6. Projections

5. Emacs functionalities

5.1. Csound-x

To do

5.1. ABC mode

To do

5.1. Csound-x

To do

5.2. Eshell

To do

5.3. Python software : Music21 and AthenaCL

To do

5.4. Clojure and the Java API for Csound

To do

25 See http://squeak.org/Documentation/

6. Supported standards and formats

ABC

MIDI

OSC

Scala

Csound

7. Bundled programs

Wavesurfer

Timidity

Sox

ABC2MIDI

Clojure

References

Surmulot home page : http://www.zogotounga.net/surmulot/surmulot.html