# Joachim Heintz
# Creating Dynamic Musical Networks with Csound

**Workshop Conservatorio Pesaro**
**1/2 April 2019**

## 01 What is this about

Sound not to be considered as a material we produce, possess and shape in any way we want.
Rather: Sound as an own being, with own desires, with reactions to other sounds, thus forming a network of sounding entities.
A network of internal and external changes, potentially infinite.

## 02 Some important properties of a sounding unit

A single "living" unit of this network has quite different properties. Some main questions:

- What is the **sound gestalt** or **body of sound**?
  In the most simple case, it can be a sine tone of a certain pitch, volume and duration.
  But it can (and usually it will) be much more complex like what we call "Gestalt".
- What is its **desire** (self-will, conatus, tendency)?
  In which context (how/when) wants the unit to appear?
  The answer (depending on our phantasy!) opens a complex field of very different possibilities.
  Some simple possibilities (from the sound's perspective):
  - I want to appear once every 30 seconds.
  - I want to appear only if ... is also there.
  - I want to appear only if ... is not there.
- What are possible **influences** on this unit?
  (In a way, dependency and irritability are main qualities of life.)
  Units can be influenced by the presence or absence of others, or one unit may force another one to appear or not to appear ... again an open field of possibilities.
- What are **reactions** to these influences?
  Musically this is probably the most important area of decisions. Anything is possible, and perhaps a reaction is not always the same — as we may accept things three times but then we explode ...
  Some simple examples are: changes of pitch, duration, volume of the sound; or pausing instead of playing; or vice versa; or starting to change the sound to another Gestalt as consequence of an influence or conflict.

## 03 Decisions and phantasies

Let us start with an example, as simple and basic as possible: Only two units, one a simple sine, the other a short sequence of filtered impulses. First step is to decide more in detail about the four main properties we just discussed:

| *Unit* | *Sound / Gestalt* | *Desire* | *Influences* | *Reactions* |
|---|---|---|---|---|
| A | sine with fadein/fadeout<br>duration 3-5 seconds<br>pitch 70-80 (midi note number)<br>volume -30 to -20 dB | sound followed by a pause of 5-8 sec | if B is playing | play one octave lower and softer |
| B | sequence of impulses<br>duration 1-2 sec<br>pitch 60-70<br>volume -12 to 0 dB | sequence followed by a pause of 5-10 sec | if A is playing | play twice as long and louder |

## 04 Implementing the sounds / gestalten

First step to implement this simple example is to code the two units only as Gestalt and Desire, without Influences and Reactions. I will use functional code style for Csound in the following. So rather than

```
   iDb random -30, -20
   kImpulsePitch random 60, 70
   aSound poscil 0.2, 400
```

I will write[1]

```
   iDb = random:i(-30,-20)
   kImpulsePitch = random:k(60,70)
   aSound = poscil:a(0.2,400)
```

Note that in this case we *always* have to specify the rate by adding *:i* or *:k* or *:a* to the opcode name.[2]

The implementation of A's gestalt is quite straightforward:

```
   iMidiNote = random:i(70,80)
   iDb = random:i(-12,-6)
   p3 = random:i(3,5)

   aSound = poscil:a(ampdb(iDb),mtof(iMidiNote))
   aOut = linen:a(aSound,p3/2,p3,p3/2)

   out(aOut,aOut)
```

1 This is mostly a matter of taste. For my own pieces I usually stick on the native Csound way, but for teaching the functional style seems to be more clear and more familiar from other programming languages.
2 For more information see http://write.flossmanuals.net/csound/i-functional-syntax/

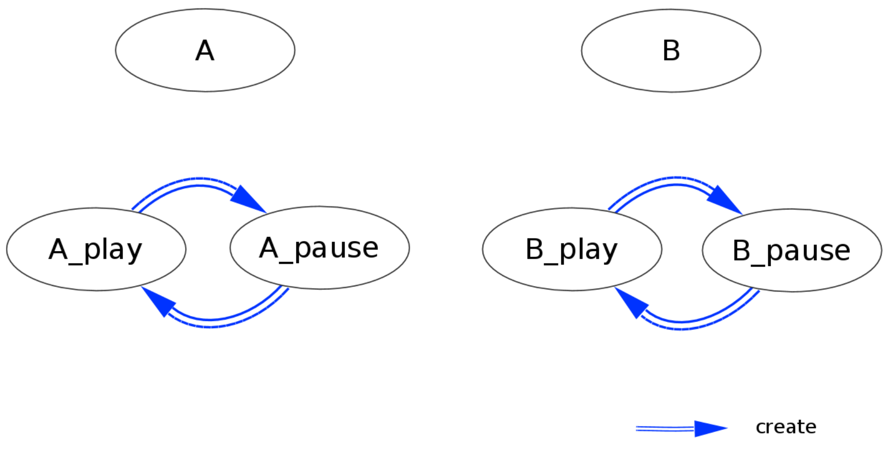
For unit B, I choose a seperation between an instrument which only plays one impulse, and another instrument which calls this for a certain time and with a certain density:

```
instr B_play

 p3 = random:i(1,2)
 iMediumFreq  = random:i(3,6)

 kTrig = metro(iMediumFreq)
 if kTrig == 1 then
  kImpulsePitch = random:k(60,70)
  kDb = random:k(-12,0)
  kOffset = random:k(0,1/iMediumFreq)
  event("i","ImpFiltered",kOffset,1,kImpulsePitch,kDb)
 endif

endin

instr ImpFiltered

 iMidiNote = p4
 iDb = p5
 iQ = random:i(1/3,2)

 aImp = mpulse:a(ampdb(iDb),p3)
 aFilt = mode:a(aImp,mtof(iMidiNote),iQ)

 out(aFilt,aFilt)

endin
```

*-> Pesaro_1.csd*

## 05 Implementing the (isolated) desire

We imagined both units to appear after a pause. There are several ways to implement this in Csound. My choice is to create an own instrument for the pause (A/B_pause) which is called by A/B_play in its last k-cycle. The pause instrument itself does the same, so that we get an infinite circle of calls, thus establishing the swaps between sounds and pauses.



```
 //start pause if last k-cycle is reached
 if(release()==1) then
  event("i","A_pause",0,1)
 endif
```

-> *Pesaro_2.csd*
EXERCISE: Change the parameters in both instruments to experience different results even at these simple sounds.

## 06 Introducing UDOs

We face now some code snippets which are mostly the same, as they are referring to typical situations of our program. This is the moment we should define them as own functions, to make the code more readable.[3] We start with a very simple one which simply wraps the event opcode to our needs:

```
opcode start, 0, S
 ;note that this is k-rate!
 S_instr xin
 event "i", S_instr, 0, 1
endop
```

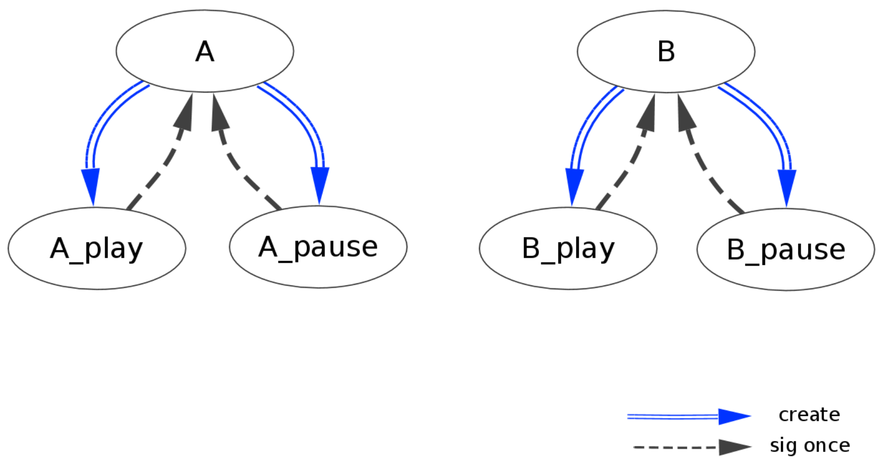
Rather than four parameters to pass to the *event* opcode, we only pass the name of the called instrument to this User Defined Opcode: *start("A_play").*
We also include the statement *-m128* in the <CsOptions> tag to reduce the printout.

-> *Pesaro_3.csd*

## 07 Make the system more sensitive

Currently the two pairs play-pause are not able to be irritated by anything. If we want them to open for any influence from "outside", we must introduce something like a sensor which senses certain parts of the reality. Again there are different possibilities to implement this into code. My solution divides the jobs between a control unit on top, which creates both, the play and the pause instrument, when time is ready for one of them.

3 Steven Yi is the master in writing *real* good (= readable) Csound code, in my opinion. See for instance https://github.com/kunstmusik/csound-live-code/blob/master/livecode.orc



In a first step we simply rebuild what we already achieved in a different way. A_play and A_pause are now not calling themselves autonomously, but send a signal to A once they finished their job, and then A calls (creates) the next instance of A_play or A_pause.

To report their state, A_play uses the software channel “A_play_done” and sends 1 at the end of its duration, otherwise 0:

```
if(release()==1) then
 chnset(k(1),"A_play_done")
else
 chnset(k(0),"A_play_done")
endif
```

(The same happens in A_pause with the channel “A_pause_done”.) Instrument A receives these signals and swaps its state in the moment it receives a 0->1 change in one of the channels:

```
//set kState whenever A_play_done or A_pause_done send a signal
if(chnget:k("A_play_done")==1 && changed(chnget:k("A_play_done"))==1) then
 kState = 0
elseif(chnget:k("A_pause_done")==1 &&
       changed(chnget:k("A_pause_done"))==1) then
 kState = 1
endif
```

Depending on this change, a new instance either of A_play (if kState==1) or A_pause (if kState==0) is created. Afterwards kState is reset to neither 0 nor 1 for listening.

```
//start play/pause depending on kState
 if(kState==1) then
  start("A_play")
 elseif(kState==0) then
  start("A_pause")
 endif

//reset state
kState = -1
```

-> *Pesaro_4.csd*
EXERCISE: Rebuild instrument B in a similar way.

## 08 Again simplification by UDOs

Sending a signal from _play or _pause to the controlling instrument once it ends is a simple thing and should read in the code in a simple way. This User Defined Opcode only requires the channel name and allows to write simply *sendSigAtEnd("my_channel_name")*:

```
opcode sendSigAtEnd, 0, Spo
 S_chn, iSendVal, iNormVal xin
 if release()==1 then
  chnset(k(iSendVal),S_chn)
 else
  chnset(k(iNormVal),S_chn)
 endif
endop
```

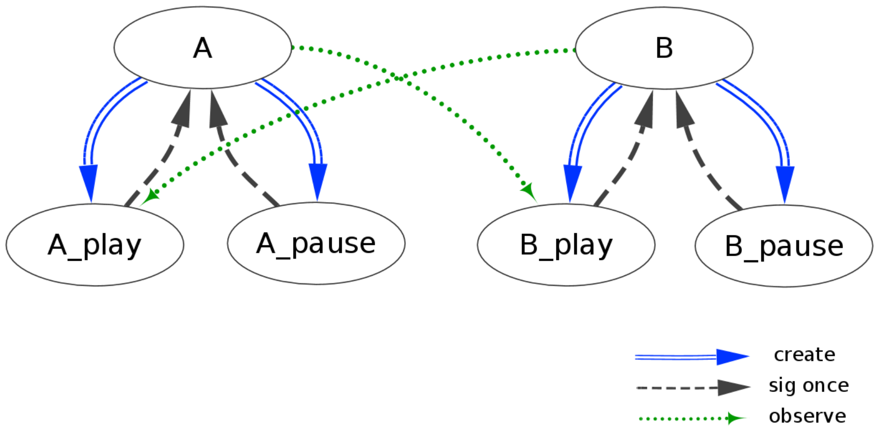
On the controller's side, the procedure of receiving a signal only once, in the moment it has changed from 0 to 1, should be wrapped in a UDO, too:

```
opcode recvSigOnce, k, Spo
 ;only outputs iSendVal if S_chn has changed from 0 to 1
 S_chn, iSendVal, iNormVal xin
 if changed2:k(chnget:k(S_chn))==1 && chnget:k(S_chn)==1 then
  kVal = iSendVal
 else
  kVal = iNormVal
 endif
 xout kVal
endop
```

-> *Pesaro_5.csd*
EXERCISE: Again apply these changes to instrument B, too.

## 09 Implementing the network

We are ready now to implement what we decided as simple example: If B is already playing, A will play one octave lower and softer; if A is playing, B will play twice as long and louder. The network now looks like this:



A has to observe whether B_play is active or not. This is very simple in Csound with the *active* opcode which returns the running number of instances of an instrument. This number we pass in the call to the A_play instrument:

```
 //get information whether B_play is active or not
 kB_active = active:k("B_play")

 //start play/pause depending on kState
  if(kState==1) then
   start("A_play",kB_active)
  elseif(kState==0) then
   start("A_pause")
  endif
```

Note that the *start* UDO is here able to send an additional parameter. We made this possible by overloading its definition:

```
opcode start, 0, S
 ;note that this is k-rate!
 S_instr xin
 event "i", S_instr, 0, 1
endop

opcode start, 0, Sk
 ;note that this is k-rate!
 S_instr, k_p4 xin
 event "i", S_instr, 0, 1, k_p4
endop
```

The first definition was already there; the second was added. Csound will take the appropriate version depending on the number of arguments.

The A_play instrument which produces the sound will receive the state as fourth input parameter, or in Csound convention: *p4*. If this parameter is 0, it will not change anything. Otherwise, it subtracts 12 from the MIDI Note Number and 10 from the dB level:

```
 //if not alone play softer and one octave lower
 if p4 > 0 then
  puts "A_play: B_play is active!", 1
  iMidiNote -= 12
  iDb -= 10
 endif
```

-> *Pesaro_6.csd*
EXERCISE: Add the same for B. Then run both instruments (A starting at 0, B starting at 5), listen and watch the output.

## 10 Results and ways ahead

We wanted to code a simple example for a dynamic network in Csound. So we cannot expect the sounding output to be very exciting. But I believe it has a big potential. Before we discuss some of the possibilities, let us look back to what we used and learned in Csound.

- We used User Defined Opcodes to simplify the code.
- We used functional syntax as alternative to traditional Csound syntax.
- We dynamically created instrument instances via the *event* opcode.
- We reported states at the last k-cycle of an instrument via then *release* opcode.
- We used *chnset/chnget* to pass messages between instruments.
- We used the *active* opcode to know about another instrument.
- We overloaded a UDO definition for more flexibility.
- We used the *random* opcode to get structures with continuous varieties.

What are now ways to continue with this model? I suggest some possibilities as exercises:

1. Add a third unit. Adjust the parameters of all three units (volume, pitch, duration, pause) so that they can live with each other and give a meaningful ensemble.
2. Define other reactions and consequences. Think about reactions which in itself are a process. For instance: Play the pitch a certain intervall lower is something is the case. This will lead to a falling pitch, so probably you have to define a minimum pitch. If this is reached, ...
3. Go deeper with the sound itself. Even with a sine tone, you can apply vibrato, glissando, different envelopes, panning, ...
4. Choose quite different sounds and see if you can work with them in the frame of this model.