Article

A model for interactive media authoring

Jean-Michaël Celerier [1,†,‡], Myriam Desainte-Catherine [1,‡] and Bernard Serpette [2,*]

[1] Affiliation 1; e-mail@e-mail.com
[2] Affiliation 2; e-mail@e-mail.com
* Correspondence: e-mail@e-mail.com; Tel.: +x-xxx-xxx-xxxx
† Current address: Affiliation 3
‡ These authors contributed equally to this work.

Academic Editor: name
Version October 6, 2017 submitted to Appl. Sci.

Featured Application: Authors are encouraged to provide a concise description of the specific application or a potential application of the work. This section is not mandatory.

Abstract: A single paragraph of about 200 words maximum. For research articles, abstracts should give a pertinent overview of the work. We strongly encourage authors to use the following style of structured abstracts, but without headings: 1) Background: Place the question addressed in a broad context and highlight the purpose of the study; 2) Methods: Describe briefly the main methods or treatments applied; 3) Results: Summarize the article's main findings; and 4) Conclusion: Indicate the main conclusions or interpretations. The abstract should be an objective representation of the article, it must not contain results which are not presented and substantiated in the main text and should not exaggerate the main conclusions.

Keywords: interactive scores; intermedia; dataflow; patcher; i-score

---

1. Introduction

Many music software fit in one of three categories: sequencers, patchers, and textual programming environments. Sequencers are used to describe temporal behaviours: an audio clip plays after another, while an automation curve changes an audio filter. Patchers are more commonly used to describe invariants: for instance specific audio filters, or compositional patterns.

We propose in this paper a method that combines the sequencer and the patcher paradigm in a live system.

The general approach is as follows: we first introduce a minimal model of the data we are operating on: namely, remote software or hardware such as OSC peripherals and sound cards. Then, two structures are presented: the first is a temporal structure, which allows to position events and processes relatively to each other, hierarchically, and in a timely fashion. The second is a graph structure akin to dataflows. This graph uses special connection types to take into account the fact that nodes of the graph might not always be active at the same time. Both structures are then combined: the state of the temporal processes is bound to the dataflow nodes. This combination is then expanded with specific implicit cases that are relevant in computer music workflows. These cases are described using structures wrapping the temporal and dataflow graphs.

We compare the various models in the context of music creation: what entails using only the temporal structure, only the graph structure, and the combination of both.

The latter model is shown to have enough expressive power to allow for recreation of common audio software logic within it: for instance traditional or looping audio sequencers. Additionnally, its use is presented in sample compositions: the first one is an example of audio editing, the second an interactive musical installation.

1.1. State of the art

There is a long-standing interest in the handling of time in programming languages, which is intrinsically linked to how the language handles dynamicity.

PEARL90[**?**][1] provides temporal primitives allowing for instance to perform loops at a given rate for a given amount of time. More recently, Céu has been introduced as a synchronous language with temporal operators, and applications to multimedia[**?**].

OpenMusic is a visual environment which allows to write music by functional composition. It has been recently extended with timed sequences allowing to specify evolutions of parameters in time[**?**].

Likewise, the Bach library for Max [**?**] allows to define temporal variations of parameters during the playing of a note by with the mechanism of slots. The processes controlled by such parameters are then available to use in the Max patch.

The Max for Live extension to Ableton Live allows to embed Max patches in the Ableton Live sequencer. Through the API provided, one can control the execution of various elements of the sequencer in Max; automations in Live can also be used to send data to Max patches at a given time.

A method for dynamic patching of Max abstractions based on CommonLisp has been proposed by Thomas Hummel[**?**] to reduce resource usage by enabling and disabling sub-patches at different points in the execution of a program. This has the advantage of saving computing power for the active elements of the score.

Dataflows and especially synchronous dataflows have seen tremendous usage in the music and signal processing community. A list of patterns commonly used when developing dataflow-based music software is presented in [**?**]. Formal semantics are given in [**?**]. Specific implementation aspects of dataflow systems are discussed in the Handbook of Signal Processing Systems[**?**].

Dynamicity in dataflows is generally separated in two independent aspects: dynamicity of the data, and of the topology. The first relates to the variability on the streams of tokens, while the second is about changes to the structure of the graph. Boolean parametric dataflows[**?**] have been proposed to solve dynamicity of topology, by introducing conditionals at the edges.

base: max, pd, séquenceurs: cubase/protools , live/bitwig...

openmusic

antescofo

inscore

1.2. Context of this research

This paper follows existing research on interactive scores, as part of the i-score project. Previous research focused on operational semantics for interactive scores, based on time automatas[] or Petri nets[], mainly for software verification purposes. In contrast, we give here domain-centered functional semantics which models the current C++ implementation of the software.

[**?**]

2. Orchestrated data

We first define the data we operate on. External devices are modeled as a tree of optional parameters.

---

[1] Not to be mistaken with the Perl language commonly used for text processing

Value parameters can have values of common data types such as integer, float, etc. Audio parameters are arrays that contain either the current input audio buffers of the sound card or the buffers that will be written to the sound card's output.

The tree of nodes is akin to the methods and containers described in the OSC specification.

$$\begin{aligned}
\textbf{Value} &= \text{Float} \mid \text{Int} \mid \text{Bool} \mid \text{String} \mid \ldots \\
\textbf{ValueParameter} &= \text{Value} \times \text{Protocol} \\
\textbf{AudioParameter} &= \text{Float}[][] \times \text{Protocol} \\
\textbf{Parameter} &= \text{ValueParameter} \mid \text{AudioParameter} \\
\textbf{Node} &= \text{String} \times \text{Maybe Parameter} \times \text{Node}[]
\end{aligned}$$

Parameters and nodes bear additional metadata which is not relevant to describe here: textual description, tags, etc.

The parameters's associated values match the state of an external device: synthesizer, etc. Multiple protocols are implemented to allow this: for instance OSC, MIDI, etc.

We define two core operations on parameters:

$$\begin{aligned}
\textbf{pull} &: \text{Parameter} \to \text{Parameter} \\
&\quad (v, p) \mapsto (v', p) \text{ where v' is the current value of the remote device} \\
\textbf{push} &: \text{Parameter} \times \text{Value} \to \text{Parameter} \\
&\quad (v, p), v' \mapsto (v', p) \text{ and v' is sent to the remote device}
\end{aligned}$$

3. Temporal model

The temporal model is two-fold: it is a hierarchical tree of processes, whose durations and execution times are directed by intervals. The beginning and end of the intervals is subjected to various conditions that will be presented ; these conditions allow for various interactive behaviours. Processes are any kind of relevant artistic process: automations, notes sequences, sound files. In particular, specific dispositions of intervals and conditions are implemented as processes themselves, scenario and loop, which allows for hierarchy.

We note: TC for temporal conditions, IC for instantaneous conditions, I for intervals. First, these elements are defined, then the semantics imposed on them by the scenario and the loop are presented. These semantics allow both serial and parallel execution.

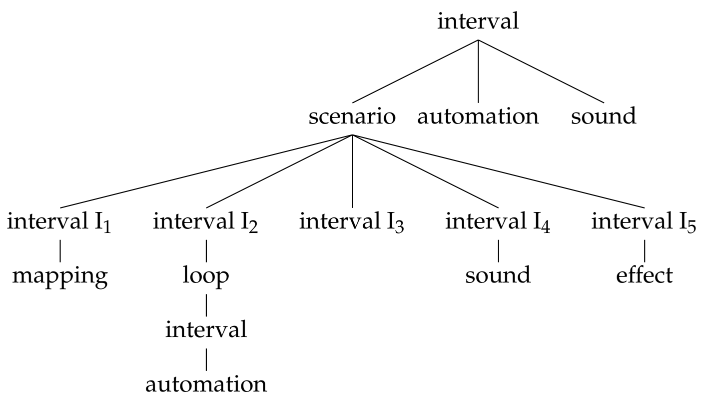
3.1. Data types

3.1.1. Conditions and expressions

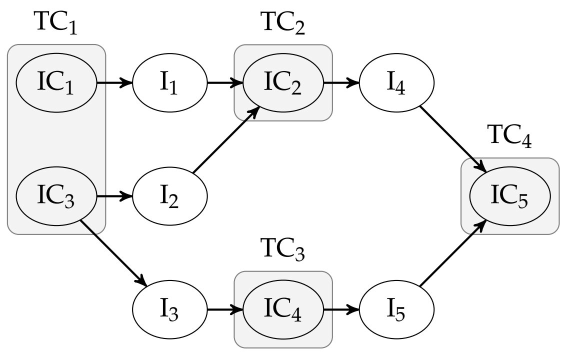
We first define the conditional operations we want to be able to express. We restrain ourselves to simple propositional logic operands: **and**, **or**, **not**.

Expressions operate on addresses and values of the device tree presented in chap. **??**, according to the grammar in **??**.

Formally, expressions are defined as a tree: Let **Comparator** be an identifier for standard value comparison operations: $<, \leq, >, \geq, =, \neq$ and **Operator** standard logical operators **and** & **or**.



(a) Hierarchical tree



(b) Temporal DAG

$$
\begin{aligned}
\textbf{Atom} &: (\text{Parameter} \mid \text{Value}) \times (\text{Parameter} \mid \text{Value}) \times \text{Comparator} \\
\textbf{Negation} &: \text{Expression} \\
\textbf{Composition} &: \text{Expression} \times \text{Expression} \times \text{Operator} \\
\textbf{Impulse} &: \text{Parameter} \times \text{Bool} \\
\textbf{Expression} &: \text{Atom} \mid \text{Negation} \mid \text{Composition} \mid \text{Impulse}
\end{aligned}
$$

Two operations are defined on expressions and the data types that compose them:

- **update** : Expression → Expression. Used to reset any internal state and query up-to-date values for the expressions. For instance, **update** on an **Atom** fetches if possible new values for the parameters, why may include network requests.

  Precisely:

$$
\begin{cases}
\text{update}: & \text{Composition} \rightarrow \text{Composition} \\
 & (e_1, e_2, o) \mapsto (\text{update } e_1, \text{update } e_2, o) \\
\text{update}: & \text{Negation} \rightarrow \text{Negation} \\
 & e_1 \mapsto \text{update } e_1 \\
\text{update}: & \text{Atom} \rightarrow \text{Atom} \\
 & \begin{cases} (\text{parameter } p_1, \text{parameter } p_2, o) \mapsto (\text{pull } p_1, \text{pull } p_2, o) \\ (\text{parameter } p_1, \text{value } v_2, o) \mapsto (\text{pull } p_1, v_2, o) \\ \ldots \end{cases} \\
\text{update}: & \text{Impulse} \rightarrow \text{Impulse} \\
 & (p, b) \mapsto (p, \text{false})
\end{cases}
$$

- **evaluate** : Expression → Bool. Performs the actual logical expression evaluation, according to the expected logical rules.

- An atom is a comparison between two parameters, a parameter and a value, or two values.
- Negations and compositions are the traditional predicate logic building blocks.
- We introduce a specific operator, "impulse", which allows to decide whether a value was received.

3.1.2. Interval

We want to be able to express the passing of time, for a given duration. This duration may or may not be finite.

A duration is defined as a positive integer. An interval is at its core a set of durations: a min, an optional max, and the current position. The lack of max means infinity. An interval is said to be fixed when its min equals its max. It may be enabled or disabled.

$$
\begin{aligned}
\textbf{Status} &= \text{Waiting} \mid \text{Pending} \mid \text{Happened} \mid \text{Disposed} \\
\textbf{Interval} &= \text{Duration} \times \text{Maybe Duration} \times \text{Duration} \times \text{Status}
\end{aligned}
$$

The time scale is not specified by the system: for instance, when working with audio data it may be better to use the audio sample as a base unit of time. But many applications don't use the audio rate: when working purely with visuals it may be better to use the screen refresh rate as time base in order not to waste computer resources and energy.

3.1.3. Instantaneous condition

Then, we want to be able to enable or disable events and intervals according to a condition, given in the expression language seen in **??**. An instantaneous condition is defined as follows:

$$\textbf{Condition} = \text{Expression} \times \text{Interval}[] \times \text{Interval}[] \times \text{Status}$$

It is preceded and followed by a set of intervals.

Expressions are disabled either when they are false or when they are preceded by a non-null number of intervals, all of them already disabled through other conditions. This propagates recursiveley to the following intervals and conditions.

3.1.4. Temporal condition

A temporal condition is used to synchronize starts and ends of intervals, while allowing to implement behaviours such as : "start part *B* when the fader is at 0".

Asynchronicity: because if in a given tick we receive the successive messages: false, true, false, we want to be able to trigger even if the "last seen" message is "false". Thus the condition evaluation operates asynchronously; however, the actual triggering is synchronous.

3.1.5. Operations

```
type process =
NodeProcess of nodeProcess | Scenario of scenario | Loop of loop
and interval = {
minDuration:duration;
maxDuration : duration option;
nominalDuration : duration;
itvStatus: status;
processes: process list
}
and condition = {
condExpr: expression;
previousItv: interval list;
nextItv: interval list;
status: status;
}
and temporalCondition = {
syncExpr: expression;
conds: condition list
}
and scenario = {
intervals: interval list ;
triggers: temporalCondition list;
}
and loop = {
pattern: interval;
startTrig: temporalCondition;
endTrig: temporalCondition;
};;
```

```
add_process interval proc: interval * proc -> interval
 (t1, t2, p, t3) -> (t1, t2, proc::p, t3)
```

```
add_event tc ic: TemporalCond * InstCond -> TemporalCond
 (..., ics, ...) -> (..., ic::ics, ...)
```

exécution :

interval:

On retourne une fonction ... graph_fun va être appliquée au graph. Faut-il avoir une liste explicite ou bien juste passer le graph et demander à chaque fonction d'appliquer ses fonctions enfants ? Le second fait plus fonctionnel mais le premier laisse moins de marge d'erreur (DRY)

```
get_node graph node_id -> node
update_node graph node_id node -> graph

graph_fun: graph -> graph ; va transformer un noeud du graphe d'une maniere donnee

tuple_first tpls: retourne les premiers elements d'une liste de paires
tuple_second tpls: retourne les premiers elements d'une liste de paires

tick: itv, count, offset : interval * duration * duration -> interval * graph_fun[]
  ((..., nom, t, pos, procs), new_date) -> (
  let procs = map procs (state _ t offset) in
    (..., t + count, t + count / nom, tuple_first procs ),
    fun (node_date,node_offset) -> (t+count, offset) :: tuple_second procs)
```

processes:

```
  state: process * t -> process * graph_fun

  described for each process (polymorphic)
```

3.2. Temporal graph: scenario

3.2.1. Creational operations

```
add_interval sc itv sev eev
```

```
add_sync sc
```

3.2.2. Execution operations

```
process_event:
```

```
make_happen:
```

```
make_dispose:
```

```
scenario_state : scenario -> scenario * state
```

3.3. Loop

Pbq: not introducing cycles in the temporal graph

```
process_event:

make_happen:

make_dispose:

loop_state:
```

4. Data model

=> set date => set offset pour offset audio (p-ê pas nécessaire si on fait comme LAStream)

5. Data graph

Questions: * node ordering * port definitions

```

(* ports *)
type edgeType = Glutton | Strict | Delayed ;;
type edge = { edgeId: int; source: int; sink: int; edgeType: edgeType; }
and audioPort = { audioPortId: int; audioPortAddr: audioParameter option; audioEdges: e
and valuePort = { valuePortId: int; valuePortAddr: valueParameter option; valueEdges: e
and port = AudioPort of audioPort | ValuePort of valuePort
;;
```

5.1. Structures

```

(* curves *)
type curve = (float * float) list ;;
let value_at curve x = 0.0;;

(* some processes *)
type automation = valuePort * curve;;
type mapping = valuePort * valuePort * curve;;
type sound = audioPort * float array array;;
type passthrough = audioPort * valuePort * audioPort * valuePort;;

type dataNode = Automation of automation | Mapping of mapping | Sound of sound | Passth
type grNode = { nodeId: int; data: dataNode; enabled: bool; date: duration; position: p


type graph = { nodes: grNode list ; edges: edge list; };;

let next_id lst f = 1 + (List.fold_left max 0 (List.map f lst));;
let next_node_id lst = next_id lst (fun n -> n.nodeId);;
let next_edge_id lst = next_id lst (fun n -> n.edgeId);;

let create_audio_port = { audioPortId = 0; audioPortAddr = None; audioEdges = []; } ;;
let create_value_port = { valuePortId = 0; valuePortAddr = None; valueEdges = []; } ;;
let test_edge = { edgeId = 33; source = 4; sink = 5; edgeType = Glutton; };;
let some_sound_data = Array.make 2 (Array.make 8 0.);;
let some_sound = Sound (create_audio_port, some_sound_data);;
```

```
let some_passthrough = Passthrough ( create_audio_port , create_value_port , create_audio

(* test *)
let test_node_1 = { nodeId = 1; data = some_sound; enabled = false ; date = 0; position
let test_node_2 = { nodeId = 34; data = some_sound; enabled = false ; date = 0; position
next_node_id [ test_node_1; test_node_2 ] ;;

let create_graph = { nodes = []; edges = [] };;
let add_node gr nodeDat =
let new_id = next_node_id gr.nodes in
let newNodeDat = match nodeDat with
| Automation a -> nodeDat
| Mapping m -> nodeDat
| Sound s -> nodeDat
| Passthrough p -> nodeDat
in
let new_node = { nodeId = new_id; data = newNodeDat; enabled = false ; date = 0; positio
(new_node, {nodes = new_node::gr.nodes; edges = gr.edges})
;;
let add_edge gr src snk t =
let new_id = next_edge_id gr.edges in
let new_edge = { edgeId = new_id; source = src; sink = snk; edgeType = t } in
(new_edge, { nodes = gr.nodes; edges = new_edge::gr.edges })
;;

(* test *)
let test_g = create_graph;;
let (snd1, test_g) = add_node test_g some_sound;;
let (snd2, test_g) = add_node test_g some_sound;;
let (p1, test_g) = add_node test_g some_passthrough;;

(* let (e1, test_g) = add_edge snd1. *)


type nodeProcess = {
node: int;
curTime: duration;
curOffset: duration;
curPos: position;
};;






node: enabled * executed * time * position * inlets * outlets * priority
add_node graph
connect graph node node edge
```

```
enable graph node
disable graph node
```

5.2. Operations

Input mix on each port

```
copy_from_global
copy_from_local
init_node
teardown_node
```

5.3. Tick description

General flow:
disable strict nodes
sort remaining nodes according to the custom order chosen (default, temporal, custom)
priority: * explicit cables * local or global address
do a tick:

```
let clear_outputs n =
  (_, ..., (map n.outputs (match p with
  | value -> clear value
  | audio -> clear audio
))

let pull_port p : port -> port

init_value : port -> value
let init_value value_port =
  if !empty value_port.cables
    mix (pull_port value_port.cables)
  pull value_port.address


let init_node g n =
  (_, (map n.inputs (match p with
    | value -> pull value
  )), ...)
exec_node:
   in
  let copy_inputs n =
  let init_node n =
    copy_inputs clear_outputs n
  in

  new_node, new_local_state = exec_node init_node n
  replace g n new_node

tick_graph:
  while: ! empty nodes
```

5.4. Data nodes

5.4.1. Passthrough

-> used for scenario and interval -> mixing at the input

5.4.2. Automation

Curves

start point + set of (segment * breakpoint)
curve + message output port $x \in [0; 1]$ -> in the nominal duration of the parent time interval.

```
state_autom :
```

5.4.3. Mapping

message input port + curve + message output port

```
state_mapping :
```

5.4.4. JavaScript

n message input port + curve + n message output port

```
state_js :
```

5.4.5. Piano Roll

notes + midi output port

```
state_midi :
```

5.4.6. Sound file

sound data + midi output port

```
state_sndfile :
```

5.4.7. Buffer

Used to keep audio input in memory
Why isn't the delay cable not enough ? can't go backwards. pb: pauser au milieu: coupure. cas dans les boucles: on réécrit par dessus (buffer vidé sur start).

```
state_buffer :
```

6. Combined model

-> on ajoute node aux tc
-> nodeprocess fait le lien entre graphnode et time process, permet de faire l'activation et l'écoulement du temps
-> offset nécessaire pour tc pour gérer l'audio (mais pourrait être ajouté dans le modèle de base. Ou bien passer une paire de pointeurs.. ?)

Figure 2. General data flow for a tick

6.1. Combined tick

Exécution complète d'un tick: Copy audio buffers and input data, execute the temporal tick, execute the graph tick, copy the output audio buffer and apply the produced state by pushing the values.

Pour être propre, il faudrait faire un "pull" général au début...

7. Proposed sequencer behaviour

Conditions et cie: The most common case for an expression is to be true.

UI: création automatique de liens implicites des enfants vers les parents => "cable créé par défaut" quand on rajoute un processus dont on marque l'entrée

=> pour toute contrainte, pour tout scénario, créer noeud qui fait le mixage => création d'objets récursivement, etc

- Problème des states dans scénario ? => states du scénario: comment interviennent-ils ? faire un scénario fantôme

- Mettre l'accent sur la recréation de la sémantique de i-score à partir du graphe: => messages: actuellement "peu" typés ; rajouter type de l'unité ?

=> pbq du multicanal: pour l'instant non traitée, on ne gère que les cas mono / stereo pour le upmix / downmix Choix pour multicanal: faire comme jamoma avec objets tilde => sliders et dispatching de canaux ? => cables: rubberband ? il faut mettre un rubberband dès qu'on a une entrée et une sortie qui n'ont pas la même vitesse relative. Dire que pour les automations ça interpole de manière naturelle avec le ralentissement et l'accélération (on sépare vitesse et granularité)

- Dire qu'on pourrait affiner en combinant plus précisément les "sous-ticks" temporels et de données pour que par exemple la production d'un état dans un scénario entraîne une condition dans un autre scénario

8. Applications and examples

8.1. Reconstructing existing paradigms

In this part we give example of reconstruction of standard audio software behaviours with the given model.

8.1.1. Audio sequencer

Notable software in this category includes Steinberg Cubase, Avid Pro Tools, . . .

The common metaphor for audio sequencers is the track, inspired from mixing desks and tape recorders. We will take the example of audio and midi tracks. Such an audio sequencer can be modeled by :

- A root: an infinite interval.
- This interval contains two processes: a scenario and an effect bus. The sound output of the scenario goes to the input of the effect bus.
- The scenario contains the actual tracks.
- These tracks are also modeled by infinite constraints.

We divide the tracks in two categories. Audio tracks are built with :

- A scenario with a single sequence of intervals, some of which may bear sound file processes and others being empty.

- An effect bus process. The output of the scenario goes to the input of the effect bus. Generally, this effect bus would end by channel operations such as panning and volume adjustment, in a similar fashion to mixing desks.

Midi tracks are built with :

- A scenario with a single sequence of intervals, some of which may bear MIDI notes processes and others being empty.
- An instrument process, which takes MIDI data and outputs sound.
- Like before, an effect bus applied to the instrument's output.

This can easily be extended with further features: sends, automations, etc.

8.1.2. Looping audio sequencer

More recently, a different kind of sequencer has emerged: the looping, non-linear sequencer. The prime exemple of this is Ableton Live. We give the example for a simplified model of live-looping without quantization.

These sequencers are also organized in tracks ; however, within a track, the musician can choose a single loop that is currently playing, and regularly switch the current loop.

Hence, the general organization stays the same than for the audio sequencer: most importantly, the way effect buses are applied does not change.

- Each clip of a track is given an index.
- Each track also has a parameter which is the next clip to play, next_clip. These parameters can be introduced as variables in the device tree.
- We replace the scenarios containing the actual sound files or midi notes by loop processes.
- The loops processes are defined with and ending temporal condition.
- Inside the loop pattern, there is a single scenario process. This scenario process has a set of parallel intervals, each with one sound file. Every interval begins with an instantaneous condition that compares the next_clip parameter to the current clip's index. Hence, at most one clip is playing at the same time in each track. If the next_clip does not change the track keeps looping on the sound file.

Extension: par ex. dans une boucle on peut mettre un autre scénario. Pb : tic qui manque. On peut y remédier en exécutant le trigger "en avance".

8.1.3. Patcher

8.2. Musical examples

8.2.1. Audio compositing

-> on utilise un scénario qui lit des parties d'une entrée son dans différents bus d'effets. L'effet peut se déclencher en retard.

Org:

Intervalle racine

Process 1: Audio input Process 2 : Scenario -> Trois itv ; entrée reliée strict à sortie de audio input ; sortie dans parent

Process 3 : FX Process 4 : scenario Audio Input -> itv 1,2,3 -> scenario -> fx -> scenario

8.2.2. Musical carousel

We present here a real-world interactive music example: the musical carousel. Each seat on the carousel has different instrument-like input devices: reactive pads, motion sensors, etc. A run in the carousel generally operates as follows:

- The first few seconds, the rules of the carousel are explained to the participants.
- The song starts: the passengers can start interacting with their instruments. An overall music is generated from their interpretation. Played notes stay in predefinite scales which may vary over time; pre-recorded parts can also be layered on top. The overall song structure can vary according to the intensity of the played music: for instance, if everyone plays *piano*, different instruments may become available in the next section of the song, a part may be shorter or longer, etc. Such variations are written by the composer for each song.
- At the end of the song, the participants hear a summarized version of the song they just played. This version also has additional corrections and adjustments applied algorithmically.

- Réutilisation des données d'entrée: scores sur certaines parties ; réutilisation de certaines notes et des pics d'intensité -> nécessite d/dx - Gammes: filtrage global du MIDI In

8.3. Notes on implementation

=> "third gen" audio sequencer. first gen: cubase, etc second gen: non-linear: ableton, bitwig third gen: entirely interactive: i-score, iannix. what else ?

reproducibilité: code source dispo

9. Evaluation and Discussion

Enforcing graph constraints: mostly done through UI. For instance: ic are created on tc, etc. No "going back" which would break DAG-ness.

Faire parenthèse sur domain driven design sur logiciels de musique qui fournit de meilleurs résultats que application directe de modèles existants (petri, etc). Peut-être donner un méta-modèle qui correspond à nos structures ?

Dire pourquoi un tic est introduit lors d'une interaction (notamment, permet de ne pas avoir de "boucle infinie" si on a une boucle de durée 0 avec deux triggers vrais) ; est aussi plus cohérent pour les utilisateurs pour qui une interaction doit être manifeste.

Avantage: manipulation uniforme des processus, que ce soit des automations, des groupes, des fichiers sons, etc.

10. Conclusion

missing: quantification

missing: sound speed

Supplementary Materials: The following are available online at www.mdpi.com/link, Figure S1: title, Table S1: title, Video S1: title.

Acknowledgments: Blue Yeti, ANRT, SCRIME All sources of funding of the study should be disclosed. Please clearly indicate grants that you have received in support of your research work. Clearly state if you received funds for covering the costs to publish in open access.

Author Contributions: For research articles with several authors, a short paragraph specifying their individual contributions must be provided. The following statements should be used "X.X. and Y.Y. conceived and designed the experiments; X.X. performed the experiments; X.X. and Y.Y. analyzed the data; W.W. contributed reagents/materials/analysis tools; Y.Y. wrote the paper." Authorship must be limited to those who have contributed substantially to the work reported.

Conflicts of Interest: The authors declare no conflict of interest. The founding sponsors had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript, and in the decision to publish the results.