Article
A model for interactive media authoring

Jean-Michaël Celerier [1,†,‡] iD **, 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.

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

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

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

---

12   **1. Introduction**

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

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

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

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

30   The latter model is shown to have enough expressive power to allow for recreation of common
31   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][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[2].

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[3].

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 [4]. Formal semantics are given in [5]. 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[6] 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[7] or Petri nets[8], mainly for software verification purposes. In contrast, we give here domain-centered functional semantics which models the current C++ implementation of the software.

We first define the temporal model, then extend it with a distinct data model which reads and produces the various inputs & outputs of the system. Then, we introduce implicit operations and defaults in the context of a GUI software to create, modify, and playback such scores. These operations allow to simplify the usage of the paradigm for composers. Real-world examples are provided and discussed.

[?]

---

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

2. Proposed sequencer behaviour

We give here an overview of the whole system ; the choices and ideas proposed will be explained in detail in the following sections.

The overall goal is to associate traditional dataflow graphs with temporal semantics. In terms of model, three structures are present:

- A global environment in the form of a tree of parameters. It represents external software and hardware such as sound cards, MIDI controllers or OSC-compliant software
- The dataflow graph which will generate and process musical data
- The temporal graph which will specify when and how the dataflow graph runs.

For the dataflow graph, the traditional musical programming patterns of and are used.

The temporal graph allows three things :

- Embedding interaction choices in the time-line.
- Arbitrary hierarchy.
- Merging of loop-based and timeline-based control: we show in section 10.1 that this is enough to allow both time-based and loop-based behaviors to co-exist in a single structure and user interface, unlike existing approaches which splits those in two mostly distinct domains ; this enables a large array of possible intermediary behaviors.

Then, at each tick, the temporal graph runs as described in section 4. This produces tokens in the dataflow graph nodes. Once tokens have been produced for every temporal structure, the data graph runs. Graph nodes which did not receive any token for a given tick will not be executed.

To accommodate for the temporal semantics, dataflow semantics are extended with :

- The ability to specify input and output addresses to ports. This allows nodes to read and write directly from the global environment, in a specified way and can be used to leverage type information associated with the parameters.
- Special connection types between edges to leverage the fact that not all nodes may be running at the same time.

When the graph runs, nodes read and write from their input and output ports ; relevant values are then copied in other ports or in the environment by the supervising algorithm.

The environment is separated in two parts: a local part which can serve to reuse the result of given computations in further node executions, and a global part which maps to messages sent to external devices.

For the sake of simplicity, the user interface merges the temporal and the data graphs in a single view. Section **??** presents in detail some automatic set-up and enforced assumptions done by the software.
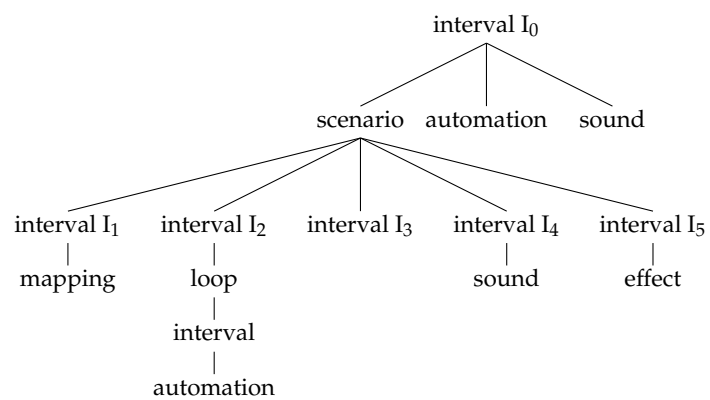
3. Orchestrated data

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

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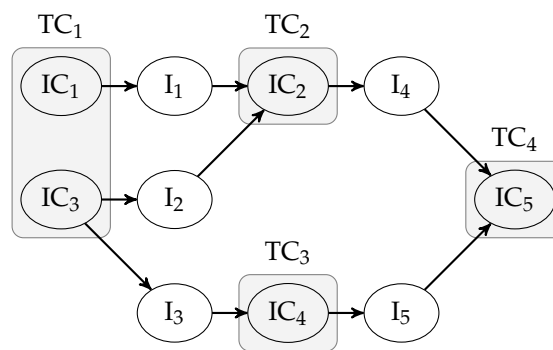
The tree of nodes is akin to the methods and containers described in the OSC specification.

[cite control-signal separation]

[cite audio processing in buffers]

interval I_0

scenario   automation   sound

interval I_1 interval I_2 interval I_3 interval I_4 interval I_5

mapping   loop   sound   effect

interval

automation

(a) Hierarchical tree

TC_1 TC_2

IC_1 → I_1 → IC_2 → I_4

TC_4

IC_3 → I_2

IC_5

TC_3

I_3 → IC_4 → I_5

(b) Temporal DAG corresponding to the scenario

$$\textbf{Value} = \text{Float} \mid \text{Int} \mid \text{Bool} \mid \text{String} \mid \dots$$
$$\textbf{ValueParameter} = \text{Value} \times \text{Protocol}$$
$$\textbf{AudioParameter} = \text{Float}[][] \times \text{Protocol}$$
$$\textbf{Parameter} = \text{ValueParameter} \mid \text{AudioParameter}$$
$$\textbf{Node} = \text{String} \times \text{Maybe Parameter} \times \text{Node}[]$$

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:

$$\textbf{pull} : \text{Parameter} \rightarrow \text{Parameter}$$
$$(v, p) \mapsto (v', p) \text{ where v' is the last known value in the remote device}$$
$$\textbf{push} : \text{Parameter} \times \text{Value} \rightarrow \text{Parameter}$$
$$(v, p), v' \mapsto (v', p) \text{ and v' is sent to the remote device}$$

4. Temporal model

The temporal model is twofold: it is a hierarchical tree of processes, whose durations and execution times are directed by intervals organized in a Directed Acyclic Graph (DAG). The beginning and end of the intervals is subjected to various conditions that will be presented ; these conditions allow for various interactive behaviors. Processes represent various kinds of relevant artistic process: automations, notes sequences, sound files ; they are unrelated to the process concept in operating systems. 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, P for processes. 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 of musical processes.

4.1. Data types

4.1.1. 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 3.

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**.

```
type subexpr = Var of string | Value of value;;
type expression =
| Greater    of subexpr*subexpr
| GreaterEq  of subexpr*subexpr
| Lower      of subexpr*subexpr
| LowerEq    of subexpr*subexpr
| Equal      of subexpr*subexpr
| Different  of subexpr*subexpr
| Negation   of expression
| And        of expression*expression
| Or         of expression*expression
| Impulse    of impulseId*string
;;
```

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

152 • An atom is a comparison between two parameters, a parameter and a value, or two values.
153 • Negations and compositions are the traditional propositional calculus building blocks.
154 • We introduce a specific operator, "impulse", which allows to decide whether a message was
155   received for a given variable.

4.1.2. Temporal processes

157 Temporal processes are executed by intervals at each tick. The actual processes will be defined in
158 sections , 4.3, 4.4, 5.

A process is a type P associated with the following operations:

$$\textbf{start} : \text{P} \rightarrow \text{P}$$
$$\textbf{stop} : \text{P} \rightarrow \text{P}$$
$$\textbf{tick} : \text{P} \times \mathbb{Z} \times \mathbb{Z} \times \mathbb{R} \rightarrow P$$
$$\textbf{Process} : \text{Scenario} \mid \text{Loop} \mid \ldots$$

4.1.3. Interval

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

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

```
type interval = {
  itvId: intervalId;
  itvNode: nodeId;
  minDuration: duration;
  maxDuration : duration option;
  nominalDuration : duration;
  speed: float;
  processes: process list
}
```

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

4.1.4. Instantaneous condition

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

```
type condition = {
  icId: instCondId;
  condExpr: expression;
  previousItv: intervalId list;
  nextItv: intervalId list;
}
```

187 It is preceded and followed by a set of intervals.

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

191 4.1.5. Temporal condition

192 A temporal condition is used to synchronize starts and ends of intervals, while allowing to
193 implement behaviors such as : "start the chorus when the fader is at 0".

194 Temporal conditions carry instantaneous conditions, which will be evaluated at the moment
195 where the temporal condition becomes true.

```
196 type temporalCondition = {
197   tcId : tempCondId;
198   syncExpr: expression;
199   conds: condition list
200 }
```

201 *4.2. General execution*

202 The overall execution consists in walking the tree of intervals and processes being executed,
203 increasing their date, and generating tokens for the associated nodes. Executing a temporal interval
204 means :

205 1. Increase its date in the score state.
206 2. Execute the tick function of all the registered processes one after each other, with the same date
207   and offset.
208 3. Register a tick for the node associated with the interval, and collapse the functions returned from
209   the ticking of processes.

$$\text{TickInterval} : \text{Interval} \times \text{Duration} \times \text{Offset} \times \text{State} \rightarrow \text{GraphFun} \times \text{State}$$

210 The type of a tick function for any given process is given in **??**; we now consider processes with a
211 non-trivial temporal behavior: scenario and loop.

212 *4.3. Temporal graph: scenario*

213 The central point in the execution of scenarios is the execution of the temporal conditions.

214 The overall progression in the scenario occurs like this:

215 1. Check for any root of the scenario that may start ; try to process them.
216 2. Increase the date of every running time interval up to their max (if any), in the capacity allowed
217   by the remaining time in the tick.
218 3. Check for finished temporal conditions, process them, and potentially propagate remaining parts
219   of the tick to following time intervals.
220 4. Return to step 2. as long as any time interval can execute.

221 4.3.1. Execution of a temporal condition

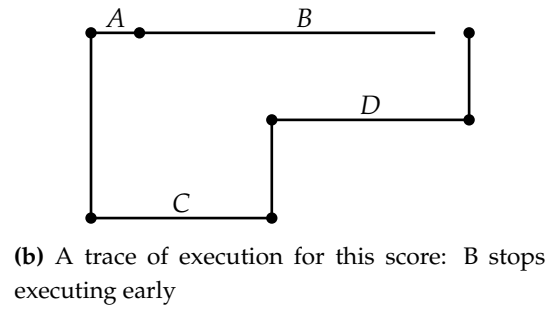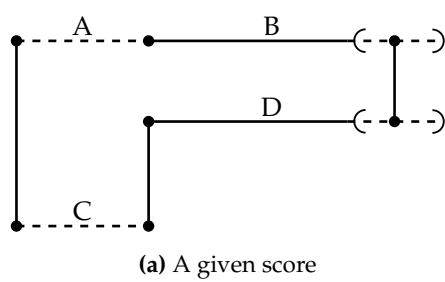222 First, let us define the following concepts for instantaneous conditions: they can be in different
223 states:

224 • **Waiting**: the execution area has not been reached yet.
225 • **Pending**: the execution area has been reached.
226 • **Happened**: has been executed successfully.
227 • **Disposed**: has not been executed.

228 The execution area, for an instantaneous condition, is the part where all the previous active time
229 intervals have reached their min duration.

230 The evaluation of a temporal condition's expression is performed at each tick, as soon as all
231 instantaneous conditions are pending.

232 Executing an instantaneous condition means checking its expression.

233 • If it is true: the previous intervals are stopped, the next intervals are started, and it is marked as
234   happened.
235 • Else, the previous intervals are stopped, and it is marked as disposed.

(a) A given score

(b) A trace of execution for this score: B stops executing early

Figure 2. A problematic case. A and C can have different durations during the execution, but the end of B and D are synchronized. In this case, the choice made is to stop executing the intervals that reached their max duration, and keep executing the others until their reach their min. The other alternative would be to keep the max constraint.

236 4.3.2. Potentially incoherent programs and their solutions

237

238 1.

239 4.3.3. General execution algorithm

240 *4.4. Loop*

241 Pbq: not introducing cycles in the temporal graph

242 **5. Data model**

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

244 **6. Data graph**

245 *6.1. Structures*

```
246  type edgeId = EdgeId of int;;
247  type nodeId = NodeId of int;;
248  type portId = PortId of int;;
249  type edgeType =
250      Glutton
251    | Strict
252    | DelayedGlutton of value list
253    | DelayedStrict of value list
254  ;;
255  type edge = {
256      edgeId: edgeId;
257      source: portId;
258      sink: portId;
259      edgeType: edgeType;
260  };;
261  type port = {
262      portId: portId;
263      portAddr: string option;
264      portEdges: edgeId list;
265  };;
266  type token_request = {
267    token_date: duration;
268    position: position;
269    offset: duration;
270  };;
271  type dataNode =
272      Automation of automation
```

```
273      | Mapping of mapping
274      | Sound of sound
275      | Passthrough of passthrough ;;
276  type grNode = {
277        nodeId : nodeId ;
278        data : dataNode ;
279  };;
280  type graph = {
281        nodes : grNode list ;
282        edges : edge list ;
283  };;
284  type grNodeState = {
285      executed : bool ;
286      prev_date : duration ;
287      tokens : token_request list
288  };;
289  type graph_state = {
290        node_state : (nodeId * grNodeState) list ;
291        port_state : (portId * value option) list
292  };;
```

6.2. Operations

Input mix on each port

```
295  let init_port (p:port) g gs (e:environment) =
296
297  let init_node n g gs (e:environment) =
298
299  let write_port p (g:graph) (gs:graph_state) (e:environment) =
300
301  let teardown_node n g gs e =
```

6.3. 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:

```
309  let rec sub_tick graph gs nodes (e:environment) =
```

6.4. Data nodes

Say that in the C++ code, the port kinds are statically checked : no way to mistake input from output.

```
313  type curve = float -> float ;;
314  type automation = port * curve ;;
315  type mapping = port * port * curve ;;
316  type sound = port * float array array ;;
317  type passthrough = port * port ;;
```

6.4.1. Passthrough

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

- Automation: start point + set of (segment * breakpoint)

  tween

  curve + message output port $x \in [0; 1]$ -> in the nominal duration of the parent time interval.
- Mapping: message input port + curve + message output port
- Javascript: n message input port + curve + n message output port
- Piano roll: notes + midi output port

- Sound file: sound data + midi output port
- Passthrough:
- Buffer: Used to keep audio input in memory
- Metronome:
- Constant: writes a value at each tick 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).
- Shader

7. 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.. ?)

7.1. Combined tick and general flow

Exécution complète d'un tick: Copy audio buffers and input data, execute the temporal tick, apply the function to the graph state, 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...

```
let add_tick_to_node nodeId token (gs:graph_state) =
;;
```

Multiple possibilities for a tradeoff between accuracy and performance:

- Use the requested ticks. This allows to have a better the performance - latency ratio at the expense of sample-accuracy for the control data.
- Use the requested ticks and tick all the nodes at the smallest granularity: given two nodes A, B, with tokens at $t = 10$, $t = 25$ for A and $t = 17$ for B, tick all the nodes at 10, 17, 25.
- Maximal accuracy : tick with a granularity of one sample every time.

```
let main_loop root graph duration granularity
                    (state:score_state) ext_events ext_modifications =
```

Détailler l'approche réactive

7.2. Details

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 si demandé (propriété de l'ui "propagate"). N'a de sens que pour l'audio de par la nature homogène de ces flux. => "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)

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

374 **8. Audio behaviour**
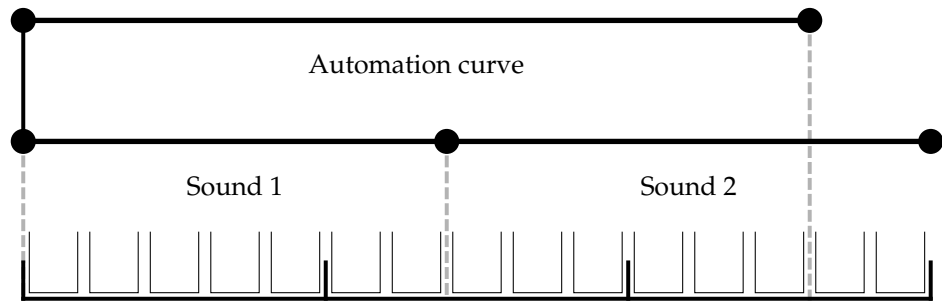
Figure 3. A scenario. The small bins represent individual audio samples ; the bigger bins represent the tick rate of the sound card. For the sake of the example, one can assume that the automation curve is used to control the output volume of the englobing scenario, not represented here.

Table 1. Value of token requests for the scenario 3. The tokens are written as (previous date → current date, offset).

	Start	Tick 1	Tick 2	Tick 3
Automation	$(0 \to 0, 0)$			
Sound 1	$(0 \to 0, 0)$	$(0 \to 5, 0)$	$(5 \to 7, 0)$	
Sound 2			$(0 \to 3, 2)$	$(3 \to 8, 0)$
Scenario	$(0 \to 0, 0)$	$(0 \to 5, 0)$	$(5 \to 10, 0)$	$(10 \to 15, 0)$

375

;
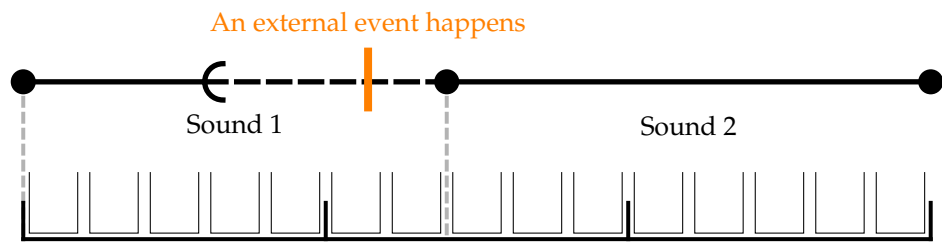
**Figure 4.** A scenario with an interaction.

376
377
378 Expliquer tick avec données, offset, etc.
379 Cas complexe:
380 dernier tick d'une boucle qui a un enfant fichier son + automation

381 **9. Mapping from visual language**

382 All the objects in the visual language correspond to the objects presented earlier, at the exception
383 of the states.
384 States : intervals with a duration equal to zero. IC / TC / interval / processes : no change edges :
385 between processes

Table 2. Value of token requests for the scenario 4.

	Start	Tick 1	Tick 2	Tick 3
Sound 1	$(0 \to 0, 0)$	$(0 \to 5, 0)$	$(5 \to 10, 0)$	
Sound 2				$(0 \to 5, 0)$
Scenario	$(0 \to 0, 0)$	$(0 \to 5, 0)$	$(5 \to 10, 0)$	$(10 \to 15, 0)$
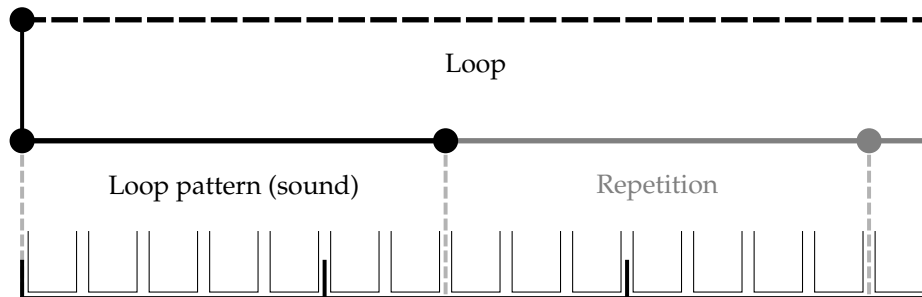
Figure 5. A loop. The small bins represent individual audio samples ; the bigger bins represent the tick rate of the sound card. For the sake of the example, one can assume that the automation curve is used to control the output volume of the englobing scenario, not represented here.

10. Applications and examples

10.1. Reconstructing existing paradigms

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

10.1.1. 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.

Table 3. Value of token requests for the scenario 5

	Start	Tick 1	Tick 2	Tick 3
Pattern	$(0 \to 0, 0)$	$(0 \to 5, 0)$	$(5 \to 7, 0), (0 \to 3, 2)$	$(3 \to 7, 0), (0 \to 1, 4)$
Loop	$(0 \to 0, 0)$	$(0 \to 5, 0)$	$(5 \to 10, 0)$	$(10 \to 15, 0)$

10.1.2. Looping audio sequencer

More recently, a different kind of sequencer has emerged: the looping, non-linear sequencer. The prime example 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".

10.1.3. Patcher

10.2. Musical examples

10.2.1. 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

10.2.2. 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 -> mapping - Gammes: filtrage global du MIDI In => on le met dans des addresses

- Addresses utilisées en global: /tempo, /scale, /quantification

10.3. 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

type-safety de l'écriture des nœuds

réimplémentation de la séparation controle / input par dessus.

algorithme qui séquence les contrôles

messages: sample accurate ; par tick, on ordonnance pour un nœud donné

11. 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.

12. Conclusion

missing: quantification

missing: sound speed

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.

1. Halang, W.A.; Pereira, C.E.; Frigeri, A.H. Safe object oriented programming of distributed real time systems in PEARL. Proceedings of ISORC-2001. IEEE, 2001, pp. 87–94.

Version November 29, 2017 submitted to *Appl. Sci.*

15 of 15

2.   Santos, R.C.; Lima, G.F.; Sant'Anna, F.; Rodriguez, N. CÉU-MEDIA: Local Inter-Media Synchronization Using CÉU. Proceedings of the 22Nd Brazilian Symposium on Multimedia and the Web; ACM: New York, NY, USA, 2016; Webmedia '16, pp. 143–150.

3.   Garcia, J.; Bouche, D.; Bresson, J. Timed Sequences: A Framework for Computer-Aided Composition with Temporal Structures. TENOR2017; , 2017.

4.   Arumí, P.; García, D.; Amatriain, X. A dataflow pattern catalog for sound and music computing. Proceedings of PLOP06. ACM, 2006, p. 26.

5.   Benveniste, A.; Caspi, P.; Le Guernic, P.; Halbwachs, N. Data-flow synchronous languages. Workshop/School/Symposium of the REX Project (Research and Education in Concurrent Systems). Springer, pp. 1–45.

6.   Bempelis, E. Boolean Parametric Data Flow Modeling-Analyses-Implementation. PhD thesis, Université Grenoble Alpes, 2015.

7.   Arias, J.; Celerier, J.M.; Desainte-Catherine, M. Authoring and automatic verification of interactive multimedia scores. *Journal of New Music Research* **2016**, pp. 1–19.

8.   Allombert, A.; Assayag, G.; Desainte-Catherine, M.; others. A system of interactive scores based on Petri nets. Proc. of the 4th Sound and Music Conference, 2007, pp. 158–165.