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.

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

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.

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.

[**?**]

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.

cite control-signal separation

cite audio processing in buffers

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 section10.1 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.

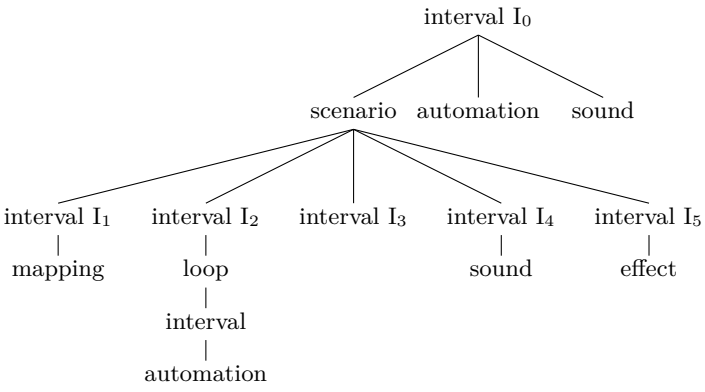
To accommodate for the temporal semantics, dataflow semantics are extended with :

- 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.

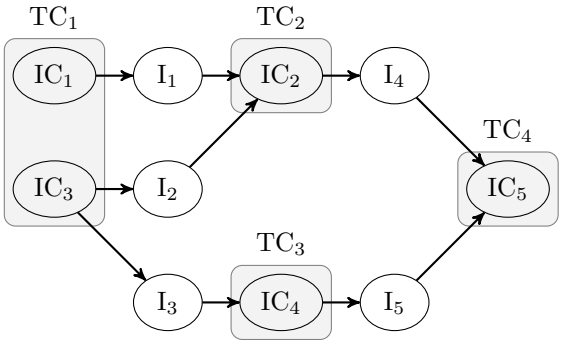
When the graph runs, nodes read and write from their input and output ports ; relevant values are then copied in other ports or in the environment by the supervising algorithm.

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.



(a) Hierarchical tree



(b) Temporal DAG corresponding to the scenario

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.

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

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:

pull : Parameter $\to$ Parameter

$(v, p) \mapsto (v', p)$ where v' is the last known value in the remote device

push : Parameter $\times$ Value $\to$ Parameter

$(v, p), v' \mapsto (v', p)$ 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
;;
```

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

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

4.1.2 Temporal processes

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

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

$$\begin{aligned} \textbf{start} &: \mathrm{P} \to \mathrm{P} \\ \textbf{stop} &: \mathrm{P} \to \mathrm{P} \\ \textbf{tick} &: \mathrm{P} \times \mathbb{Z} \times \mathbb{Z} \times \mathbb{R} \to P \\ \textbf{Process} &: \text{Scenario} \mid \text{Loop} \mid \dots \end{aligned}$$

4.1.3 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.

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

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 artistic 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.

4.1.4 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:

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

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 recursively to the following intervals and conditions.

4.1.5 Temporal condition

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

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

```
type temporalCondition = {
  tcId: tempCondId;
  syncExpr: expression;
  conds: condition list
}
```

4.2 General execution

The overall execution consists in walking the tree of intervals and processes being executed, increasing their date, and generating tokens for the associated nodes. Multiples tokens can be generated for a single data node in a given tick; the use case will be explained in section 4.4. Executing a temporal interval means :

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

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

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

4.3 Temporal graph: scenario

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

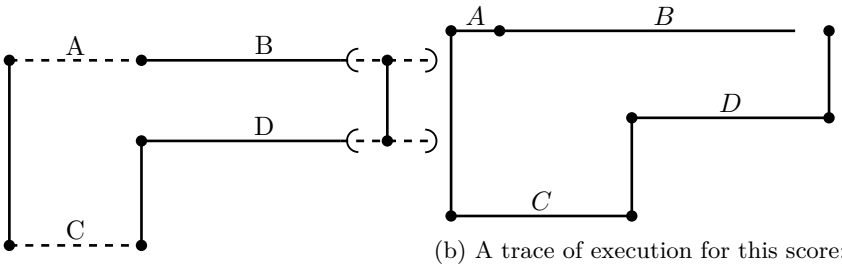
The overall progression in the scenario occurs like this:

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

4.3.1 Execution of a temporal condition

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

- **Waiting**: the execution area has not been reached yet.
- **Pending**: the execution area has been reached.
- **Happened**: has been executed successfully.
- **Disposed**: has not been executed.



(a) A given score

(b) A trace of execution for this score: B stops executing early since its max was reached

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.

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

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

Executing an instantaneous condition means checking its expression.

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

4.3.2 Potentially incoherent programs and their solutions

4.3.3 General execution algorithm

* Main score and sub-scores: main score only triggers once. * Convenience: Sub-scores can always re-trigger and be restarted.

Expliquer comment on évite les deadlocks: notamment, on progresse toujours dans les autres contraintes même si on arrive au max

4.4 Loop

The first idea for loops would be to create a cycle in the scenario graph presented earlier. However, acyclicity of this graph is a desirable property: this allows to keep the complexity of graph traversal operations used lower, which matters in a real-time execution context.

Hence, we chose instead to model the loop as its own process with simpler semantics: a single time interval will loop. Since intervals can contain other processes, we can still encounter complex musical behaviour in it: whole scenarios can loop.

A loop is simply defined as :

```
type loop = {
  itv: interval
  startTc: temporalCondition;
  endTc: temporalCondition;
}
```

At each tick of the loop, the interval is itself executed. If either of the temporal conditions is interactive, we follow the same process than for the scenario: we wait until the next tick to perform the triggering instead of doing it immediately. Else, further tokens are added to the interval and its child processes, starting from zero. An example of this process is given in fig. 5.

Null loop durations are prevented at creation time: else, any tick operation would cause an infinite loop since there would never be any progression in the loop.

4.5 Conclusion

5 Data model

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

Canaux audio: comme Jamoma ; groupés (vs comme Max / Pd ou chaque canal a son cable)

6 Data graph

6.1 Structures

```
type edgeId = EdgeId of int;;
type nodeId = NodeId of int;;
type portId = PortId of int;;
type edgeType =
    Glutton
  | Strict
  | DelayedGlutton of value list
  | DelayedStrict of value list
;;
type edge = {
    edgeId: edgeId;
    source: portId;
    sink: portId;
    edgeType: edgeType;
};;
type port = {
    portId: portId;
    portAddr: string option;
    portEdges: edgeId list;
};;
type token_request = {
  token_date: duration;
  position: position;
  offset: duration;
};;
type dataNode =
    Automation of automation
  | Mapping of mapping
  | Sound of sound
```

```
  | Passthrough of passthrough;;
type grNode = {
    nodeId: nodeId;
    data: dataNode;
};;
type graph = {
    nodes: grNode list;
    edges: edge list;
};;
type grNodeState = {
  executed: bool;
  prev_date: duration;
  tokens: token_request list
};;
type graph_state = {
    node_state: (nodeId * grNodeState) list;
    port_state: (portId * value option) list
};;
```

6.2 Operations

Input mix on each port

```
let init_port (p:port) g gs (e:environment) =

let init_node n g gs (e:environment) =

let write_port p (g:graph) (gs:graph_state) (e:environment) =

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:

```
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.

```
type curve = float -> float;;
type automation = port * curve;;
type mapping = port * port * curve;;
type sound = port * float array array;;
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)

- 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 -> tout réduire à un tick

8 Audio behaviour

	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	$(0 \to 0, 0)$	$(0 \to 5, 0)$	$(5 \to 10, 0)$	$(10 \to 15, 0)$

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

;

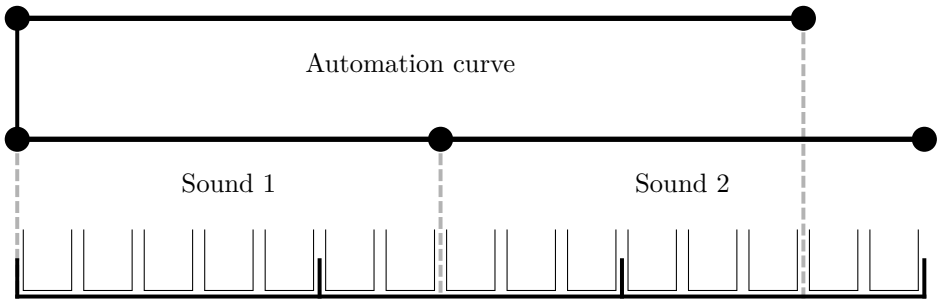


Figure 3: A scenario. The small bins represent individual audio samples ; the bigger bins represent the tick rate of the sound card. For the sake of the example, one can assume that the automation curve is used to control the output volume of the englobing scenario, not represented here.

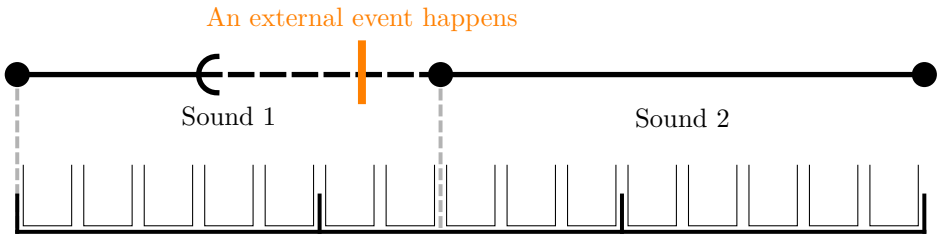


Figure 4: A scenario with an interaction.

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

9 Mapping from visual language

All the objects in the visual language correspond to the objects presented earlier, at the exception of the states.

States : intervals with a duration equal to zero. IC / TC / interval / processes : no change edges : between processes

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.

	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	$(0 \to 0, 0)$	$(0 \to 5, 0)$	$(5 \to 10, 0)$	$(10 \to 15, 0)$

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

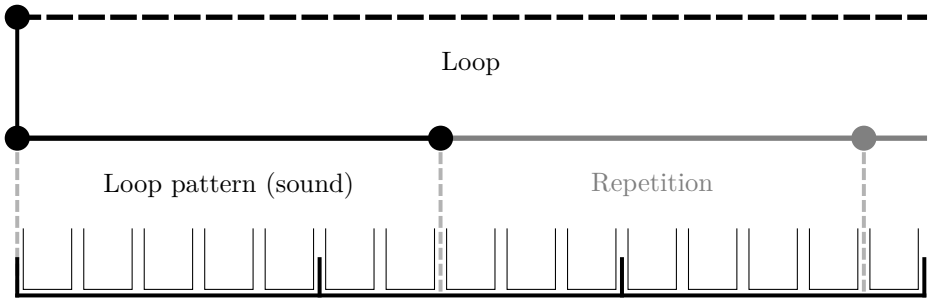


Figure 5: A loop containing a sound.

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.

	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	$(0 \to 0, 0)$	$(0 \to 5, 0)$	$(5 \to 10, 0)$	$(10 \to 15, 0)$

Table 3: Value of token requests for the scenario 5

- 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.

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