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 September 29, 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 define the temporal structure, which allows to position events and processes relatively to each other, hierarchically, and in a timely fashion. Then, we define a graph structure akin to dataflows, which is extended with 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.

The usage of the system is presented in example compositions: the first one is an example of audio editing, the second an interactive musical installation.

1.1. State of the art

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

openmusic

antescofo
inscore

1.2. Relationship with i-score

-> formalisation du papier icmc
-> refonte suite à tentative avec LibAudioStream

2. Orchestrated data

We first define the data we operate on. External devices are modeled as a tree of optional parameters ; parameters can have values of common data types such as integer, float, etc.

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:

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

3. Temporal model

We note: TC for the temporal conditions, IC for the instantaneous conditions, I for the intervals. chaining.

3.1. Data types

process, interval, event, sync

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**.

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

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

- **update** : Expression $\rightarrow$ 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 $\rightarrow$ 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. The most common case for an expression is to be true.

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. Process

3.1.6. Operations

```
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:

```
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)
```

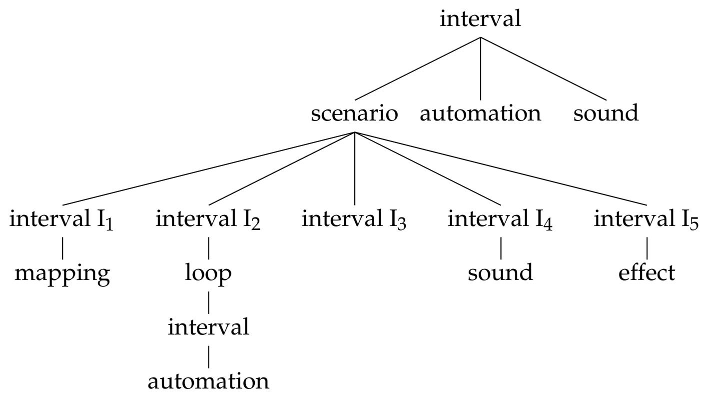

Figure 1. Hierarchical tree

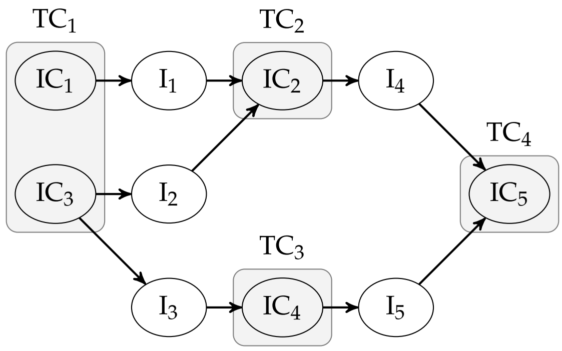

Figure 2. Temporal DAG

3.2. Temporal graph: scenario

3.2.1. Creational operations

```
add_interval sc itv sev eev
```

```
add_sync sc
```

3.2.2. Execution operations

```
state:
```

```
process_event:
```

```
make_happen:
```

```
make_dispose:
```

3.3. Loop

Pbq: not introducing cycles in the temporal graph

4. Data model

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

4.1. Data types

```
add_node graph
connect graph node node edge
```

4.2. Operations

4.3. Data graph

4.4. Data nodes

4.4.1. Passthrough

-> used for scenario and interval

4.4.2. Automation

4.4.3. Mapping

4.4.4. JavaScript

4.4.5. Piano Roll

4.4.6. Sound file

4.4.7. Sound input

4.4.8. Mix

5. Combined model

5.1.

6. Proposed sequencer behaviour

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)

Exécution complète d'un tick:

- Copie des buffers audio - Exécution du tick temporel - Récupération des states

- 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

7. Applications

- Exemple article Myriam: micro-montage et sélection d'effets - Carrousel

8. Evaluation

8.1. Notes on implementation

- Recréation séquenceur traditionnel, patcher, et ableton live (vue session).

=> "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. Discussion

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

10. Conclusion

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.