Extending dataflows with temporal graphs

Jean-Michaël Celerier, Myriam Desainte-Catherine, Jean-Michel Couturier

Extending dataflows with temporal graphs

Jean-Michaël Celerier
LaBRI, Blue Yeti
Univ. Bordeaux, LaBRI, UMR 5800,
F-33400 Talence, France.
Blue Yeti, F-17110 France.
jcelerie@labri.fr

Myriam Desainte-Catherine
LaBRI, CNRS
Univ. Bordeaux, LaBRI, UMR 5800,
F-33400 Talence, France.
CNRS, LaBRI, UMR 5800,
F-33400 Talence, France.
INRIA, F-33400 Talence, France.
myriam@labri.fr

Jean-Michel Couturier
Blue Yeti, F-17110 France.
jmc@blueyeti.fr

ABSTRACT

Numeric musical and artistic creation is in great part based on well-known patcher softwares: Max, PureData, OpenMusic, vvvv...

Yet, handling temporal evolutions and structures on large scales remains a central problem: for instance, writing a song with introduction, chorus, verse, chorus, parts in Max presents many hurdles to the composer. Hence, multiple software have introduced tracks and automations to provide a temporal control of a given set of parameters in a single or even multiple concurrent time-lines.

We present an extension to the dataflow paradigm used by patchers that allows to handle the execution of any parts of a computation in time. That is, instead of sending control messages to existing patchs, they are run in a time-line which orchestrates the data bindings between different parts of the dataflow. This enables dynamic behaviours when creating temporal compositions. An example is given with Pure Data patches and an i-score scenario.

1 Introduction

This work explores the association of macroscopic temporal semantics with the dataflow graphs generally used in music and signal processing.

We analyze the meanings that can be given to processes executing in time, when these processes have a data dependency and do not execute at exactly the same time.

For instance, given two processes p_1, p_2, what can be said of a program where p_1 executes from $t = 0$ to $t = 5$ seconds, and p_2 from $t = 3$ to $t = 6$ when p_1 and p_2 operate on the same data?

An example would be an audio filter: p_1 is a low-pass and p_2 a reverb; we want the reverb to activate only after an action of a performer and stop after a few seconds.

The simplest strategy can be to only allow execution when all processes are active and have been given a definite order of execution: in this case the overall program will only execute during the intersection of the activation of p_1 and p_2, which goes from $t = 3$ to $t = 5$. While safe from the point of view of software execution, we will show that other execution policies leveraging information from the temporal layout of the processes can provide composers with new creative capabilities, through an example that uses Pure Data.

An environment associated with the dataflow graph is introduced; it contains the values of the inputs and outputs of the processes in this graph. Then, since a graph may not always be fully active, we show that it is also meaningful to have implicit connections between nodes of the graph, that will use the environment. This allows dynamic routing of the processes according to their temporal order during execution, and extends routing to pattern-matched elements of the environment instead of single values. Three different behaviors between the input and output parameters of the dataflow nodes are presented: strict, glutton, and delayed.

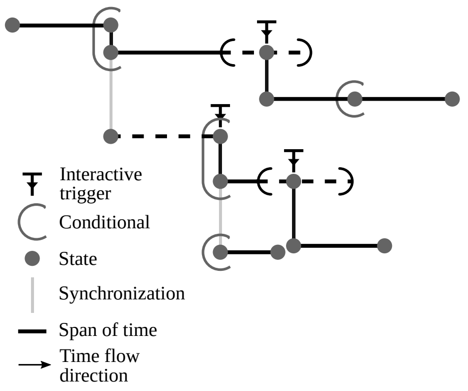


Figure 1: Part of an OSSIA scenario, showcasing the temporal syntax used. A full horizontal line means that the time must not be interrupted, while a dashed horizontal line means that the time of this time constraint can be interrupted to proceed to the following parts of the score according to an external event.

1.1 Related works

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[10] [1] provides temporal primitives allowing

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

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[13].

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[9].

Likewise, the Bach library for Max [1] 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[11] to reduce resource usage by enabling and disabling subpatches 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 [2]. Formal semantics are given in [4]. Specific implementation aspects of dataflow systems are discussed in the Handbook of Signal Processing Systems[5].

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[3] have been proposed to solve dynamicity of topology, by introducing conditionals at the edges.

1.2 Temporal formalism

This work implicitly uses the OSSIA formalism [7] in its examples to provide primitives relative to the evolution of time: time constraints (horizontal lines), time nodes (vertical lines). The elements of its syntax are presented in fig. 1.

It is important to note that any other system of temporal relationships between processes could be used instead.

2 Definitions

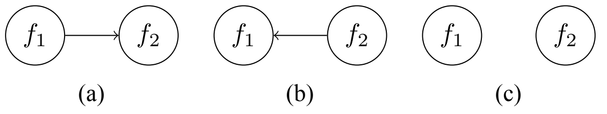


Figure 2: Possible dataflows between f_1 and f_2. As can be seen in fig. 3, there are no relationships between f_1 and f_2 in the temporal graph.

We call process an entity that consists in a function f, an associated state, any number of inputs and output ports, and an activation status. f has a single argument $t \in \mathbb{R}^+$. The execution of f may read any number of tokens from the inputs and write any number of tokens to the outputs.

The function associated to an inactive process may not be executed.

We rely on two families of graphs for defining the execution of a program:

- Synchronous dataflow graphs between processes: $G_d(V_d, E_d)$ where V_d is the set of processes and E_d the set of data bindings between processes.
- Temporal graphs, based on the OSSIA formalism: $G_t(V_t, E_t)$ where V_t is the set of time nodes and E_t is the set of time constraints in a score. For coherency, we call these graphs temporal graphs.

In the following, we consider a program consisting in a dataflow, and a temporal graph.

A program may consist in more than a single temporal graph: they are defined as processes themselves, which provides them with time information.

Each time constraint has an associated set of references to processes. A process may only be executed if it is referenced at least once in a time constraint, at the exception of a special "main" process that acts as an entrypoint.

The execution of a program is synchronous and driven by a scheduler. At each tick, the state of each temporal graph is updated with the following informations:

- Which processes are currently activated.
- For how long these processes have been activated.

This is done according to the various events that occur and may change the order of activation of processes.

Processes are then sorted according to their dependencies and activation status, and finally executed.

Throughout this paper, we will use the execution traces in fig. 4a and fig. 4b as running example, as well as the relationships between f_1 and f_2 present in fig. 2.

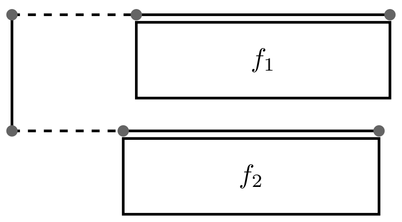
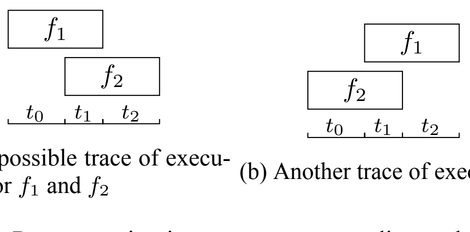


Figure 3: The example temporal graph we use: both f_1 and f_2 can start at any time. f_1 and f_2 both have a given, finite duration.



(a) A possible trace of execution for f_1 and f_2

(b) Another trace of execution

Figure 4: Process activation traces corresponding to the graph given in 3

3 Environment, implicit and explicit connections

In the data flow paradigm generally used in music software, nodes are linked together through connections in their respective input and output ports.

This work extends the data flow approach with a permanent external environment, that contains a mapping of key-value pairs. Such a pair is called a variable.

Keys are specified with OSC-like[8] addresses: `/foo/bar` and values can be any kind of data: numbers, strings, etc. The values of the environment may vary independently of the execution of the program: for instance elements of the environment may be changed asynchronously through the network, graphical widgets or physical controls.

We then have to provide a mean to use the values of this environment to the dataflow. Thus, we extend the input and output ports with the following explicitness semantic:

- An explicit port is a port that has been connected manually to another port. An input port may only read tokens from the output port it is connected to. This connection is static and will not change for the duration of the execution. [2]
- An implicit port is a port that has not been connected to another port. It may dynamically read and write to any number of addresses from the environment to perform its work.

During a tick, ports have access to three sets of values, corresponding to scopes:

- Global scope: The values that were in the environment at the beginning of the tick. These values are accessible to every process.
- Local scope: The values produced by previous implicit output ports in the dataflow. Section 4 presents the various policies that nodes can leverage to use these values. The local scope is cleared at the beginning of a tick. The values written to it are committed to the global environment at the end of the tick.
- Connection scope: the explicit scope between two ports; the data flowing from one output port can only go to input ports it is connected to. This is the explicit case.

For a given execution following fig. 2a, the evolution of such parameters would behave as in table 1.

Address	Tick	After f_1	After f_2	Tick
Global `/a`	a_0	a_0	a_0	$f2(f1(a_0))$
Local `/a`	$\emptyset$	$f_1(a_0)$	$f2(f1(a_0))$	$\emptyset$

Table 1: Evolution of a variable in the global and local scope during the execution of a tick, in the case of an implicit connection. Both f_1 and f_2 are active. "Tick" indicates the beginning of a new tick; the temporal progression is left-to-right.

This mechanism allows for new behaviors during the creation of a dataflow program.

[2] Reactive environments would of course be able to alter such connections at run-time; relevant mechanisms are outside of the scope of this paper.

4 Relationships

We are interested in the relationships between nodes of the dataflow graph when they produce compatible tokens, whether the production is specified implicitly or explicitly, and when taking into account deactivated nodes. That is, given a node of the dataflow executing and producing tokens, we must define which following nodes, if any, will receive the tokens and when will they execute.

We propose three relationships: strict, glutton, and delayed.

These relationships are expressed between ports of two nodes of the dataflow.

Two nodes may not always be explicitly connected to each other through a cable. We have to provide an order between them since execution requires a total ordering of nodes.

For this, an additional set of directed edges between nodes of the dataflow is used to provide this order. Methods to set up these edges are discussed in section 5.

4.1 Strict relationship

In a given tick, an execution of a node engaged in a strict relationship with another node depends on the other node being active.

For instance, take the case in fig. 4a where f_1 and f_2 both read from a and write to a where a is an address. We use the dataflow given in fig. 2a.

Let $\text{commit}(a, x)$ be the function that commits a value x to the address a in the local scope, and $\text{pull}(x)$ the function that reads the value of the address x from the global scope. We assume that f_1, f_2 always have access to this information.

We will get the following behaviors during each slice:

- During t_0: $\emptyset$.
- During t_1: $\text{commit}(a, (f_2 \circ f_1)(\text{pull}(a)))$.
- During t_2: $\emptyset$.

A strict relationship between two nodes may only be defined through an explicit connection.

This relationship should be used when a computation does not make sense on its own.

4.2 Glutton relationship

An execution of a node will happen even if the nodes it is connected to are not active; instead, data will be read and written from the environment.

If we take the same case than previously, the behavior is:

- t_0: $\text{commit}(a, f_1(\text{pull}(a)))$.
- t_1: $\text{commit}(a, (f_2 \circ f_1)(\text{pull}(a)))$.
- t_2: $\text{commit}(a, f_2(\text{pull}(a)))$.

This relationship may be explicit or implicit.

In the explicit case, the output of f_1 goes to the input of f_2 through a cable. If an address has been specified for each port in addition to the explicit connection:

- If f_1 is not active, f_2 reads from the local scope instead, or the global scope if the required address is not available.
- If f_2 is not active, f_1 writes to the local scope instead.

Such a behavior is conceptually similar to a guitarist's pedal board: not all pedals will always be active, but we want the signal to keep flowing even if a pedal is disabled.

4.3 Delayed relationship

A connection between an output and an input is delayed through bufferisation in a queue.

The delayed connection can behave in two ways:

- Readers of the buffer always start from the same point: the beginning of the previous function in the callback chain. The frame pointer would be located in each delayed connection, and would not be shared between processes.
- Readers of the buffer continue from the latest read position. The frame pointer would be located in the source port, and would be shared across all processes reading from it.
  This behavior can be useful when multiple functions should apply successively to a single buffer, as in fig. 6. However, it would also create concurrent accesses problems if two nodes happened to read an output at the same time.

Arumi presents in [2] the advantages and drawbacks of storing tokens at the output or input ports.

This relationship has to be specified explicitly to allow for the tokens to be buffered.

If we have a delay connection from f_1 to f_2, the first call to f_2 will use the first value that was produced by f_1.

We also define a strictness level for the delayed connection. In the strict case, a node will only be able to execute if the source has produced enough tokens.

That is, in fig. 5, f_2 would only execute during t_1.

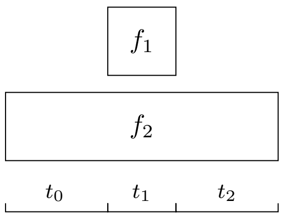


Figure 5: Another execution trace, stemming from a different score than the one presented in fig. 3. f_2's input corresponds to f_1's output, delayed: this creates a causality problem.

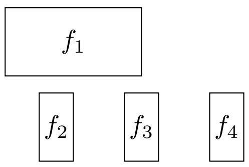


Figure 6: f_2, f_3 and f_4 read from f_1 with a delayed connection.

5 Default behaviors for orders

We assess here the various mechanisms possible to specify the dependencies between nodes of the dataflow. Information provided in the temporal graph can be leveraged to connect the nodes at runtime in a fashion that would make sense to composers & authors.

By default, any node without preceding nodes would strive to be scheduled at the earliest possible time.

Connections between ports imply a partial ordering between nodes, hence we only consider the ordering between implicitly connected nodes of the dataflow.

5.1 Manual ordering

In this case, the author explicitly specifies an order by setting up an edge between two nodes. This is the slowest process, but which gives the most precise specification.

5.2 Hierarchical ordering

We follow the hierarchical organization of the temporal graphs: each time constraint has an ordered list of processes, and each time constraint is itself ordered relative to others in a given temporal graph. This gives an order between processes, which does not require the composer to manipulate the dataflow unless a specific data connection has to be established.

5.3 Temporal ordering

In this case, the temporal order in which objects are executed becomes the order of chaining in the dataflow.

With the OSSIA semantics, this order could change at each run of a score, as showcased in fig. 7.

- t_0: f_1.
- t_1: $f_2 \circ f_1$.
- t_2: f_2.

(a) Glutton execution in the order of the fig. 4a

- t_0: f_2.
- t_1: $f_1 \circ f_2$.
- t_2: f_1.

(b) Glutton execution in the case of fig. 4b

Figure 7: Executions with a temporal ordering

6 Implementation considerations

Multiple points discussed in this article can be left to the decision of designers of dataflow systems.

6.1 Compromises in dynamicity

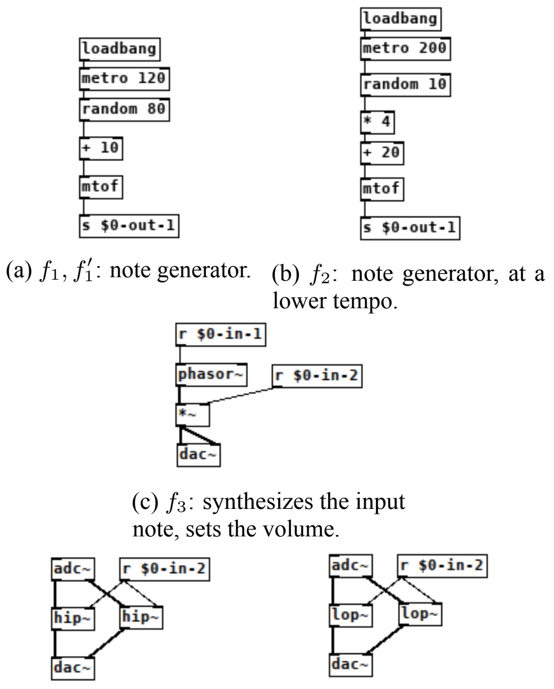
A first choice that such a designer ought to make would be the level of dynamicity of the system. For instance: would nodes of the dataflow have statically, fixed input and output addresses, or would they be defined and routed dynamically at each tick. The first case allows for better performance: more functional dependencies could be checked and set-up statically. The second enables new behaviors: for instance, a node could write a value to a random parameter at each tick to create fuzzing effects.

A second choice is the use of scopes and the behavior when a given scope does not have the required value. If an input port tries to read from an address `/a`, would the implementation allow it to read from the global scope if the value is not available in the local scope? It is up to the designer of the software to choose whether a port can work with any permutation of ports:

- When connected explicitly,
- With the local scope,
- With the global scope.

6.2 Pattern matching

More dynamic behaviors could also be achieved by allowing to use patterns instead of addresses in the port definitions.



(a) f_1, f_1': note generator.

(b) f_2: note generator, at a lower tempo.

(c) f_3: synthesizes the input note, sets the volume.

(d) f_4: high-pass filter.

(e) f_5, f_5': low-pass filter.

Figure 8: The functions we use in our example, as defined in Pure Data.

For instance, instead of accepting data from a single source, a port can specify an input pattern such as: `/foo.*/bar/b[a-zA-Z]`, as proposed in the OSC specification.
Upon execution, the input stream of the port would contain the list of values corresponding to the matching addresses.
Given a global environment with the variables:

```
/foo/bar/ba      1.0
/foo.1/bob/bu    2.0
/foo.2/bar/be   -1.0
/foo.42/bar/bo  -3.0
```

Such a pattern would yield the tokens -1.0 and -3.0 at the input of the port matching the pattern.

6.3 Default behaviors

Default behaviors can be encoded in any hierarchical unit; for instance, the global scope. Likewise, each temporal graph could also be set-up with a default connection type.
We propose a default glutton behavior; however usability studies should be made to determine the most user-friendly default.

6.4 Reference implementation

An implementation is being developed as an extension to the OSSIA library [3] and i-score software [4]. This implementation uses PureData [12] through libpd [6] to provide function execution with multiple ports with primitives useful for musical creation. As such, it allows to work with audio, midi, and messages.
The C++ implementation of the execution algorithm, provided in the OSSIA library, is not tied in any way to PureData or any other environment: PureData is used as an

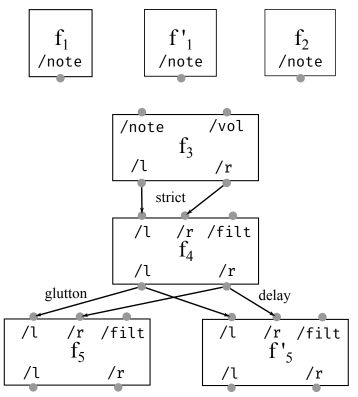


Figure 9: The dataflow relationships between instances of $f_{1,..5}$ in the program. The global environment contains variables for `/vol` and `/filt` when the execution starts. A PureData patch with similar semantics would use additionally `throw~, catch~, send, receive, delwrite~, delread~, tabwrite~, tabread~` between nodes, and UDP & OSC receivers and senders to read and write from the environment.

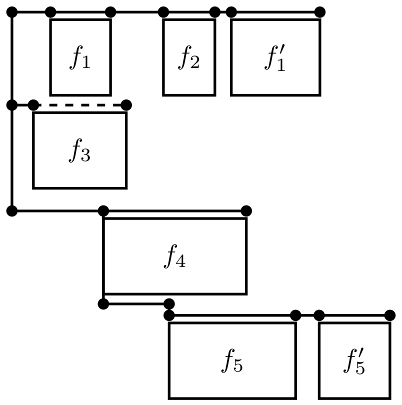


Figure 10: The temporal relationships between instances of $f_{1,..5}$ in the program.

example of implementation of a node to leverage an easy and well-known method to produce generators and filters with different kinds of inputs and outputs, which allows to cover the whole set of features. Other nodes can be implemented through inheritance of a set of C++ classes. As such, the processes currently available in i-score (automations, Javascript code, MIDI piano roll) and in i-score-audio (sound file reading, Faust and LV2 effects) are being ported to this new execution model.
In terms of user interface, PureData patches appear as nodes with their inputs and outputs in i-score. There are two views: the main view, temporal and hierarchical, which allows to position the nodes in time, draw automations, etc., and the patching view which allows to connect nodes together and set the relationship types. The temporal view instead has easily accessible text fields that allow setting input and output addresses for each port.

7 Applied example: generative music

We present here an example that leverages patches built with Pure Data[12]. Figure 8 present the Pd patches used.

[3] `https://github.com/OSSIA/libossia`
[4] `http://www.i-score.org`

Fig. 9 presents the data relationships between the Pd patches, with the mapping of inputs and outputs to addresses of the environment. The inlets and outlets are ordered so that audio ports come first and message ports come afterwards. Finally, fig. 10 presents the temporal score that orchestrates the execution of the Pd patches.

The execution happens as follows:

1. For a few milliseconds, nothing happens.
2. f_3 is started. However, there is a strict dependency between f_3 and f_4: no sound is produced; f_3 is disabled.
3. f_1 is started. Since f_3 is disabled, it writes `/note` to the environment.
4. f_1 stops, and a few milliseconds afterwards, f_4 starts, which allows f_3 to execute. f_3 uses the last value of `/note` that was put in the global scope by f_1. f_3 fetches the value of `/vol` in the environment and f_4 fetches the value of `/filt` there as well. Notice that f_3's parent time constraint is dashed: it will execute until it is manually stopped.
   The sound that can be heard is a single tone with some amount of high-pass filtering.
5. f_2 starts: random notes will be generated at a lower tempo and put at the `/note` input of f_3 directly, bypassing the environment.
6. f_5 starts: the low-pass filter adds itself after the high-pass filter.
7. f_2 stops: the sound reverts to a continuous tone (the last that was written, again).
8. f_1' starts: tones are generated again, but with a faster tempo.
9. f_4 stops: there is no sound anymore due to the strict dependency. f_5 applies its low-pass filter to silence.
10. f_5 stops, and just afterwards $f'5$ starts. Since $f'5$ was in a delayed relationship with f_4, the sound produced corresponds to the sound that was heard at the beginning of f_4, with both f_4's high pass filtering, and $f'5$'s low-pass filtering.

8 Conclusion

Throughout this paper, the effects of adding macro-temporal semantics to dataflows were studied. The first step is to introduce an environment that allows execution of a program to continue if not all the nodes of a dataflow are currently being executed. Then, we study the new possibilities offered by the presence of the environment by the different ways to interpret the absence of execution of a node. This includes matching nodes together through pattern matching, and distinct policies of execution that can take into account the evolution of the program in time.

Such global environments have been implicitly present in multiple dataflow-based software; however they are not given an explicit existence. For instance, Max externals generally handle their global state through `static` C variables. Making it explicit allows for better understanding of the execution of a program, and may enable further optimization opportunities.

After discussing the various choices left to the implementor, an example leveraging multiple PureData patches scheduled through an i-score score is explained.

A further goal for this work is to extend the definition not only to work with a single variable of time, but to any kind of activation mechanism: for instance, evolution of a position in a 3D space, etc.

The paper mostly centered around a single dataflow. A meaning has been proposed for multiple dataflows: they could be used as distinct variable scopes. If the temporal graph has hierarchical features, its use as a scoping mechanism should also be envisioned.

References

[1] Andrea Agostini and Daniele Ghisi. “A max library for musical notation and computer-aided composition”. In: *Computer Music Journal* (2015).

[2] Pau Arumí, David García, and Xavier Amatriain. “A dataflow pattern catalog for sound and music computing”. In: *Proceedings of PLOP06*. ACM. 2006, p. 26.

[3] Evangelos Bempelis. “Boolean Parametric Data Flow Modeling-Analyses-Implementation”. PhD thesis. Université Grenoble Alpes, 2015.

[4] Albert Benveniste et al. “Data-flow synchronous languages”. In: *Workshop/School/Symposium of the REX Project (Research and Education in Concurrent Systems)*. Springer, 1993, pp. 1–45.

[5] Shuvra S. Bhattacharyya et al., eds. *Handbook of Signal Processing Systems*. New York, NY: Springer New York, 2013.

[6] Peter Brinkmann et al. “Embedding pure data with libpd”. In: *Proceedings of the Pure Data Convention*. Vol. 291. Citeseer. 2011.

[7] Jean-Michaël Celerier et al. “OSSIA: Towards a unified interface for scoring time and interaction”. In: *TENOR2015*. Paris, France, May 2015.

[8] Adrian Freed and Andy Schmeder. “Features and Future of Open Sound Control version 1.1 for NIME”. In: *Proceedings of NIME13*. 2009, pp. 116–120.

[9] Jérémie Garcia, Dimitri Bouche, and Jean Bresson. “Timed Sequences: A Framework for Computer-Aided Composition with Temporal Structures”. In: *TENOR2017*. A Coruña, Spain, May 2017.

[10] Wolfgang A Halang, Carlos Eduardo Pereira, and Alceu Heinke Frigeri. “Safe object oriented programming of distributed real time systems in PEARL”. In: *Proceedings of ISORC-2001*. IEEE. 2001, pp. 87–94.

[11] Thomas Hummel. “A Common Lisp interface for dynamic patching with the IRCAM signal processing workstation”. In: *International Computer Music Conference*. 1994, pp. 216–216.

[12] Miller Puckette et al. “Pure Data: another integrated computer music environment”. In: *Proceedings of the second intercollege computer music concerts* (1996), pp. 37–41.

[13] Rodrigo C.M. Santos et al. “CÉU-MEDIA: Local Inter-Media Synchronization Using CÉU”. In: *Proceedings of the 22Nd Brazilian Symposium on Multimedia and the Web*. Webmedia ’16. New York, NY, USA: ACM, 2016, pp. 143–150.