Sonic Arts Today, Vol.10 No.1 pp.28–33

Research report

Leveraging domain-specific languages in an interactive score system

Jean-Michaël CELERIER
Université de Bordeaux - SCRIME - LaBRI - Blue Yeti

Abstract

Interactive scores have been introduced as a way to author open works which includes elements affecting the course of a performance such as conditions and interaction points. However, while the temporal control & specification aspect has been greatly detailed in the litterature, few work has been done in the ways in which the score can actually produce music or other multimedia behaviours such as lights and visual control ; the standard approach is to leverage external software through the means of network protocols such as OSC. We present here a case study of the integration of various Domain-Specific Languages (DSLs) and more general language paradigms with different specificities such as dataflow, functional and imperative, directly in the execution loop of the Ossia Score interactive score system, and their application to the authoring and real-time performance of rich audio-visual content.

1. Introduction

Time is an integral part of most multimedia processes and work, but authoring in the time domain is still an unsolved question when interactivity is involved. That is, how to model multimedia applications where the time is not fixed by an author before performance and execution, but depends on interactive actions of the application users.

We present a model and an implementation for the authoring of interactive multimedia applications based on the theory of interactive scores. In particular, this model is used to bind various DSLs together to create sonic and multimedia arts.

A starting point for interactive scores research is the interactive scores model proposed by Desainte-Catherine and Allombert in [3]. Parts of it are successors to the research done in the Jamoma project[10, 11, 12], as well as part of the French research projects Virage [1] and OSSIA [1].

We choose to rely on multiple DSLs for reasons exposed on broader computer science literature: using such languages for specific tasks has generally been recognized as efficient in comparison to using a single, general-purpose programming environment[6].

[1] http://ossia.gmea.net/

2. Interactive scores

In interactive scores, the author deliberately allows for multiple variations of a given part of the score, for instance at the note level where onsets and durations can vary, or at the scope of greater musical phrases ; this can be likened to the musical concepts of ossia and fermata. The three central questions relative to interactive scores are: “How does one write an interactive score”, “How are such scores performed”, and “How can such scores be checked for inconsistencies”. The majority of existing literature on the subject covers the two last points: we instead cover the two first ones, and in particular the question of authoring. The proposed authoring methods leverage various specific environments combined together in a single software: each environment should be used for the tasks it is specialized for in order to simplify authoring.

2.1. Interactivity

Interactivity in music, and more generally in arts has been covered by Umberto Eco in [5]: a central difference between recent open works and previous forms of art is that these works actively encourage the performer to act not only on individual parameters, but on the structure of the work itself. A way to enable this is to enumerate the possible cases that the performer will encounter, and let him choose amongst them. In [4], the question of interactivity in the context of artistic performance in the digital age is addressed: the author states that interactivity is not a “feature” that a performance may have, but instead a discrete spectrum in which the user interaction with the work is involved. The scripting of interactive pieces can be extended towards full audio-visual experiences, in the case of artistic installations, exhibitions and experimental video games.

3. The Score system

The model is twofold: a graph defines the execution of musical processes in time, and another defines the data

relationships and dependencies between the processes that will be used during a given tick of execution. In the graphical environment, the author manipulates a restricted version of both graphs which automates the creation of objects; nodes of the data graph are enabled when they start executing and disabled the rest of the time.

For instance, it is possible to apply a distortion to a sound for ten seconds, then, if a condition becomes true in the score, apply a reverberation on top of the distortion. An example of temporal graph is given in fig. 1 ; a screen-shot of the graphical environment is given in fig. 2.

Audio processing in the score engine is sample-accurate: it is possible to loop sounds perfectly, and to have precise positioning of objects in time.

The system, implemented in the Ossia Score software [2] integrates to a high level with the OSC, MIDI, and other relevant protocols, thanks to the libossia library [3] . In particular, every port of every object can be assigned an OSC address ; the port will either read or write values from this address. This also allows processing to work if not all nodes of a signal chain are active: they can fetch the relevant data from either local variables in an environment or global variables which correspond to the OSC addresses known to the system.

4. Temporal combining of relevant DSLs

The goal of this work is to assess the relevance of multiple DSLs for the sake of creating interactive multimedia scores. The first part of this work was to research embeddable environments which would be amenable to scoring in time. The following section describes the chosen DSLs.

4.1. Capacity of DSLs

Multiple features are desirable from the environments we wish to integrate with :

- Introspectability: it must be possible from C++ code to discover the parameters provided by a program in a given environment. This is to allow the creation of matching ports and controls in the host environment. We distinguish three possibilities: the environment does not support any kind of reflection, the environment allows to reflect its inputs and outputs, and the environment allows to reflect its inputs and outputs and associate semantic meaning to it: for instance, is the output meant to represent a VU-meter, a slider, etc. This third level enables automatic user interface generation.

[2] http://www.ossia.io

[3] http://ossia.github.io

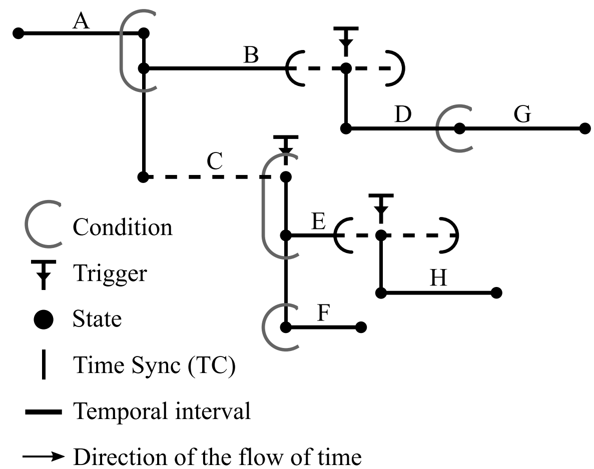

Figure 1. Part of a scenario, showcasing the temporal syntax used. A full horizontal line means that the time must not be interrupted, while a dashed horizontal line means that the time of the interval can be interrupted to continue to the next part of the score according to an external event. Execution occurs as follows: the interval A runs for a fixed duration. When it ends, an interval is evaluated: if it is false, the branch which contains B will not run. Else, after some time, the flow of time in B reaches a flexible area centered on an interaction point, also called a temporal condition. If an interaction happens, B stops and D starts. If there is none, D starts when the max bound of B is reached by the flow of time in B. Just like after A, an instantaneous condition will make G execute or not execute. In all cases, C started executing after A. C expects an interaction, without time-outs. If the interaction happens, the two instantaneous conditions which follow C are evaluated : the truth value of each will decide of the execution of E and F.

- Dynamicity: can a program in the DSL easily be reloaded and change at run-time. This enables live-coding[7] features.

- Latency and performance: is the language suitable enough for real-time execution and performance, at very small periods – typically, less than two milliseconds.

- Threading and multi-instance support: can the environment be embedded multiple times in a host software without interference between instances, and without problems if these multiple instances are called from different threads. This enables better performance at run-time.

- Relevancy: is the language famous enough that it is possible to find composers or programmers able to write programs for it easily, which is especially relevant in the context of a musical studio which does the software realisation of

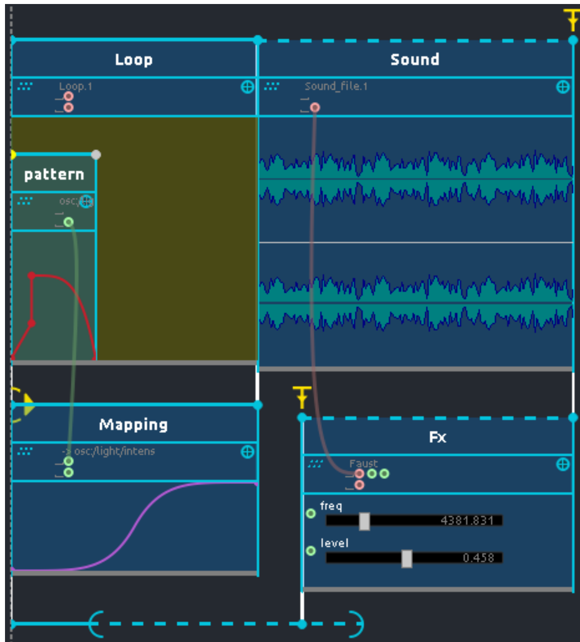

Figure 2. An example of scenario in the software. For a few seconds, an automation loops. The automation values are sent to a mapping function, which sends the resulting messages through OSC. Then, a sound plays. At some point, if an external event happens or after a few additional seconds, the sound goes through an effect built using the Faust programming language.

compositional works.

4.2. Faust

[8] is a functional programming language tailored for signal processing, able to produce efficient DSP code in a variety of environments. An example of Faust code is given in fig. 3

```
import("stdfaust.lib");

phasor(f) = f/ma.SR : (+,1.0:fmod) ~ _ ;
osc(f)    = phasor(f) * 6.28318530718 : sin;
process   = osc(hslider("freq", 440, 20, 20000, 1))
            * hslider("level", 0, 0, 1, 0.01);
```

Figure 3. Example Faust program : a sine generator with a frequency and a volume control. Courtesy of the Faust website (http://faust.grame.fr/examples/2015/09/30/oscillator.html)

Faust in particular is able to handle sample-accurate controls: it is possible to execute for any given number of samples, which allows to set precise automation values. It also provides some level of semantic information : min, max, default bounds and steps for various UI controls.

```
import QtQuick 2.0
import Score 1.0
Item {
  FloatSlider { id: in1; min: 20; max: 20000 }
  AudioOutlet { id: out1 }

  property int idx: 0;
  function onTick(oldtime, time, position, offset)
  {
    var arr = [ ];
    var n = time - oldtime;
    var freq = in1.value;
    if(n > 0) {
      var mult = 2 * Math.PI * freq / 44100;
      for(var s = 0; s < n; s++) {
        var sample = Math.sin(mult * idx++);
        sample = sample > 0 ? 1 : -1;
        sample = freq > 0 ? sample : 0;
        arr[offset + s] = 0.3 * sample;
      }
    }
    out1.setChannel(0, arr);
    out1.setChannel(1, arr);
  }
}
```

Figure 4. A sine generator written with the QML integration in the environment.

4.3. JavaScript and QML

It is sometimes more interesting to script using traditional, imperative or object-oriented programming: many scientific algorithms are for instance given under a form tailored for implementation in this kind of language. To enable this, we provide the integration of the Qt QML engine in the environment. QML is a superset language of Javascript.

An example Javascript script is provided in fig. 4. The script is a QML object. It contains child objects which will map to inputs and outputs in the system: in the example, FloatSlider and AudioOutlet. Available types are :

- Value inlets and outlets, able to send traditional data types such as integer, boolean, floating point number, string.
- Midi inlets and outlets.
- Audio inlets and outlets.
- In addition, specific types of inlets can be used for a given value kind: for instance, FloatSlider is a value inlet specialized for float types ; it will show a relevant UI widget in the user interface. Other available types are Enum (shows a list of choices), IntSlider, TextField, Toggle which all map to relevant value types. These objects are provided from our environment directly: more can be added if needed by composers.

4.4. PureData

PureData (Pd)[13] has been embeddable for a few years in host applications thanks to the work done in

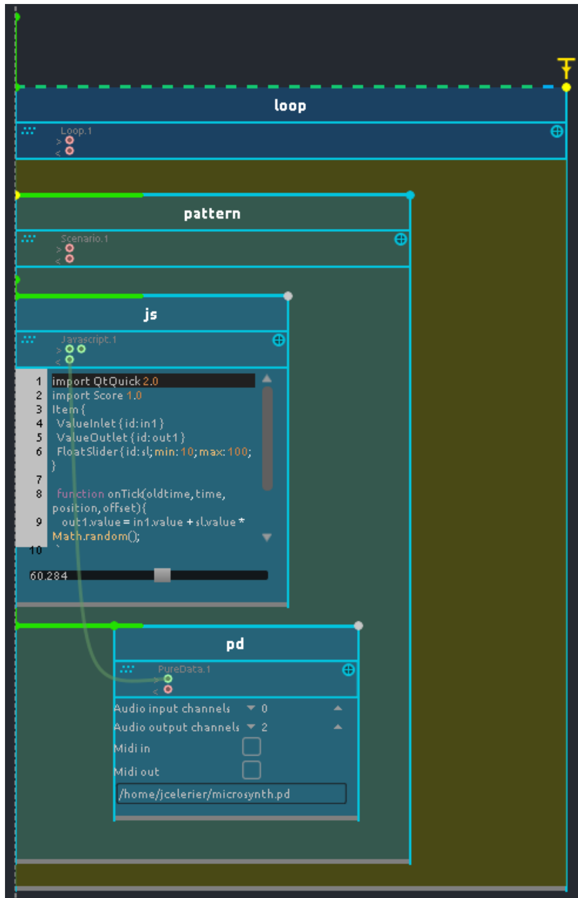

Figure 5. An example of score linking a Javascript script and a Pd patch temporally. Both scripts loop in time for a duration of a few seconds. The Javascript script gets its inputs from external OSC messages. It writes data to its output buffers ; when the Pd patch starts, it gets the values that were buffered at the beginning of the JS script, in a FIFO way.

libpd[2].

It is a prime target for our work: many composers are knowledgeable of patcher environments such as Pd and Max/MSP and as such can be easily leveraged for integration. In addition, recent work by Puckette et al. has made possible the use of multiple distinct instances of Pd in a single process which enables to have different sub-patches integrated in a score patch.

However, integration of Pd leads to a problem: it enforces a block size of 64 samples. That is, Pd will only write to its outputs every 64 audio samples. This means that sample accuracy cannot be assured with Pd: the system has to be buffered at this point.

The libpd implementation also does not provide reflection information. To enable some external control support, we scan the Pd patch save file for messages sent and received: `[r $0-mymessage]`, `[s $0-mymessage]`, `[midiin]`, `[midiout]`, `[adc~]`, `[dac~]` are used to create input and output ports in the environment.

4.5. GLSL

GLSL (GL Shading Language) is a shader language, used for generating graphics on GPUs. It can be used in the environment through a specific process, which will open a render window. In particular, we use the Interactive Shader Format [4] extension to specify inputs and outputs of the shader ; the format use a JSON-based header to specify metadata relevant to the shader which is then parsed by the environment.

In particular, this extension specifies a standard method for passing audio data to shaders in order to create interactive audio visualisations.

4.6. ExprTK

ExprTK[9] is a mathematic expression language, which follows standard mathematical conventions: for instance the expression `cos(2*x) + 1` has the expected behaviour. The main utility of ExprTK for this project is allowing to do arithmetical computations with less overhead than a whole Javascript environment.

ExprTK is currently integrated as a set of four processes in the environment:

- A value generator.
- A value mapper.
- An audio generator.
- An audio filter.

In every process, `t` gives the current time in samples, `dt` gives the time in samples since the last tick, and `pos` gives the time relative to the parent object. `x` is the input value for mapping processes. In addition, three float variables a, b, c controllable through sliders in $[0; 1]$ are provided by default to enable some controls ; more complex set-ups would be better served by the full-blown Javascript interpreter.

For instance, consider the expression `a + x * cos(t / 1000)` in the value mapper. Given the score of fig.6, and the external Pd patch of fig.7 we get the output of fig.8, split in three parts: the first one corresponds to the value of the automation mapped through the expression, the second one corresponds to the output of the OSC message `/value` and the third one correponds to the mapping of a low-frequency oscillator object. The mapping object itself writes its output to the OSC addres `/output`, which is then displayed in a Pd array.

5. DSL integration process

We present here the process used to integrate a new DSL in the system. The software consists of two part: the use interface, written in C++ with the Qt library, is based on a plug-in system: new plug-ins can be

[4] www.interactiveshaderformat.com

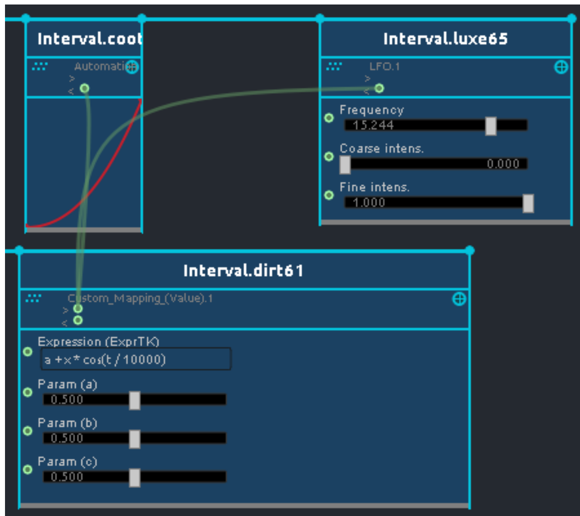

Figure 6. A score example with a simple math expression mapper. The process at the top left is an automation, the process at the top right is a low-frequency oscillator producing square waves.

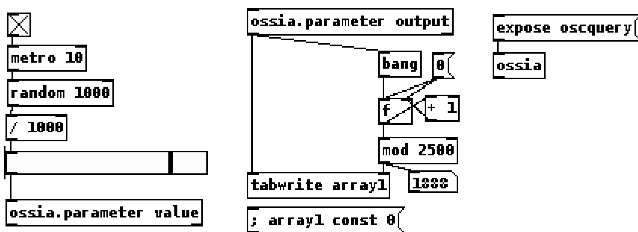

Figure 7. An external Pd patch which communicates with the score of fig.6. Objects of the libossia library are used to simplify network communication: they can be enumerated automatically over the network.

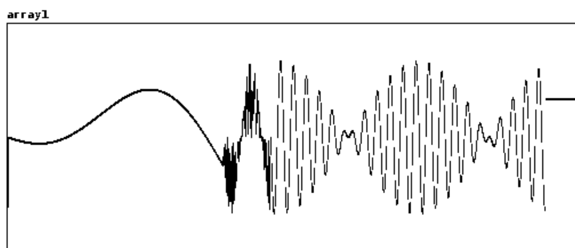

Figure 8. The array of the Pd patch in fig.7. The first slow-moving part corresponds to the automation, the second, noise-like, to the random values sent from Pd, and the third part, periodic, to the LFO object.

provided, which allow for custom UI, and specific behaviour at execution. The execution engine itself is part of the pure C++ library libossia: it can be easily leveraged from other graphical environments and is not tied to the main user interface.

The Abstract Factory design pattern is used ; objects are split in various part which each handle an aspect of the plug-in.

The method is as follows:

- Inherit from Process::ProcessModel to provide a model implementation. This class is the model on which the edition action applies, and contains the data that will be saved to disk. It should create ports according to the inputs and outputs detected from the current DSL script ; these ports will be automatically displayed in the UI. Special control ports can be created, which will show sliders in the user interface.

- Inherit from ossia::graph_node to provide an implementation of the execution object: the main function:

```
void graph_node::run(
    ossia::token_request,
    ossia::execution_state&);
```

  will be called at each tick with the number of samples and the other informations mentioned earlier: token_request is a simple structure which contains current date, current position, offset since the beginning of the current audio buffer ; execution_state allows to fetch data from network protocols. In addition, the ports of the node can of course be accessed at this point ; this is the preferred way to interact with external environments through either OSC, MIDI or other protocols.

- Inherit from Engine::Execution::ExecutionComponent to provide an implementation of the conversion between the UI model and the execution engine object. This class is where the reactive editing abilities will be implemented: in particular, if a parameter or property of the model changes during run-time, it should be updated in the execution engine. Messages are passed to the execution engine through function objects in a lock-free queue and executed at the end of an execution tick from the main execution thread. For instance, this allows hot reloading of scripts during execution while preserving atomicity if multiple actions have to be performed, in contrast with a more traditional lock-based approach which would damage performance.

- If the user interface needs special handling, for instance because a text field has to be displayed, one must reimplement Process::LayerPresenter which

handles the user interaction and propagates changes from the model to the view, Process::LayerView.

6. Conclusion

We presented a set of languages introduced in the interactive score environment Ossia Score. These languages cover various parts of the interactive installation and media creative pipeline: some are more tailored towards graphics, other towards audio, and other towards mathematical computations. These environments are rendered compatible together through standard software engineering processes ; the specificity of the environment is that it allows these various languages to be scored in time.

6.1. Future works

All the presented languages currently use run-time interpretation. While this is enough for many musical uses, a further goal would be to improve performance by using languages that are able to be compiled into efficient assembly code at run-time: this is nowadays doable with C++ code and the LLVM environment, for instance, and would reduce latency further. In particular, some environments such as Javascript may allocate memory : this is undesirable in any thread related to audio processing.

In addition, usability studies should now be performed to assess the performance of given tasks for composers with various DSLs of equivalent abilities: for instance, would more composers manage to write their pieces with Javascript scripts, Pd patches or a mix of both for their computations ?

7. Bibliography

References

[1] Pascal Baltazar et al. “Virage: Une réflexion pluridisciplinaire autour du temps dans la création numérique”. In: (2009).

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

[3] Myriam Desainte-Catherine and Antoine Allombert. “Interactive scores: A model for specifying temporal relations between interactive and static events”. In: Journal of New Music Research 34.4 (2005), pp. 361–374.

[4] Steve Dixon. Digital performance: a history of new media in theater, dance, performance art, and installation. MIT press, 2007.

[5] Umberto Eco. “The Open Work”. In: Trans. Anna Cancogini. Cambridge: Harvard University Press (1989).

[6] Paul Hudak. “Domain-specific languages”. In: Handbook of Programming Languages 3.39-60 (1997), p. 21.

[7] Thor Magnusson. “Algorithms as scores: Coding live music”. In: Leonardo Music Journal (2011), pp. 19–23.

[8] Yann Orlarey, Dominique Fober, and Stéphane Letz. “Faust: an efficient functional approach to DSP programming”. In: New Computational Paradigms for Computer Music 290 (2009).

[9] Arash Partow. The C++ Mathematical Expression Toolkit Library (ExprTk). 2000. url: https://github.com/ArashPartow/exprtk.

[10] Timothy A Place and Trond Lossius. “Jamoma: A Modular Standard for Structuring Patches in Max.” In: ICMC. 2006.

[11] Timothy Place, Trond Lossius, and Nils Peters. “A Flexible And Dynamic C++ Framework And Library For Digital Audio Signal Processing.” In: International Computer Music Conference. 2010.

[12] Timothy Place, Trond Lossius, and Nils Peters. “The jamoma audio graph layer”. In: International Conference on Digital Audio Effects (DAFx-10). 2010.

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

8. Author’s Profile

Jean-Michaël CELERIER is a french ph.d student, enrolled at the Université de Bordeaux under the direction of Myriam Desainte-Catherine. He is employed by the multimedia art company Blue Yeti for the development of the sequencer Score, and a member of the SCRIME (Studio de Création et Recherche en Informatique et Musique Électro-Acoustiques : Creation and Research Studio in Computer Science and Electroacoustic Music).

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/4.0/ or send a letter to Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.