A Web Playground for Ciaramella

Paolo Marrone
Orastron s.r.l.
Università di Udine
paolo.marrone@orastron.com

ABSTRACT

Ciaramella is a domain specific programming language for audio DSP. It experiments a fully declarative syntax and the Synchronous Data Flow computational model, featuring high modularity and composability. We implemented a web playground for coding in Ciaramella, compiling and instantly generating working web audio plugin prototypes. In order to obtain that, its compiler, Zampogna, which is written in JavaScript, has been embedded in a web page, and it has been augmented for the production of JavaScript code. We developed a simple graphical user interface targeting both domain experts and newcomers. Finally, this work lays the foundations for future WebAssembly support as target code.

1. INTRODUCTION

Ciaramella is a recent domain specific text based programming language for the description of audio DSP systems featuring an high-level of abstraction. It supports experimental approaches like declarative syntax and Synchronous Data Flow model (SDF) [1], whose combination is new to the audio programming languages field. Indeed, the existing audio programming languages, which share a similar domain with Ciaramella, do adopt different paradigms. Faust [2], for example, is purely functional; Kronos [3] is defined as a *metalanguage* and experiments interactive graphical programming and just-in-time compilation; Max Gen [4] focuses on visual programming.

In this work we explore the possibility to deploy programs written in Ciaramella as audio applications directly on the web.

The possibilities and popularity of web programming have been continuously increasing during the years. Concerning audio, W3C developed the Web Audio API [5], which is now a standard adopted and implemented by all major web browsers. Web audio provides a routing graph mechanism capable of connecting independent audio nodes, multiplexing signals and handling parameters. Some common audio nodes are provided by default, but custom ones can be user-defined. In particular, the DSP algorithm can be written in pure JavaScript or in WebAssembly (using JavaScript as glue code). The WebAssembly option allows the audio developer to use any programming language (e.g. C, C++, Rust) as far as it is supported by a compiler that targets WebAssembly. Also, it is particularly appealing for its near-native performances.

When it comes to high-level languages such as Ciaramella, Faust or Kronos, a desired web application for the end user is an online playground which permits to code, compile, instantly listen to the produced plugin, and eventually export the compiled files. Faust, Kronos and Soul, for example, have developed their own playgrounds [1] [2] [3].

Faust's compiler is written in C++ and it has been compiled to WebAssembly via the Emscripten toolchain [4] to make it embeddable in a web page[6]. Its playground comes with an auto-completion text editor, compilation, on-the-fly interactive graphical plugin creation and export features. Kronos provides a similar environment, but it is focused on a graphical composition system rather than a text editor.

The paper is organized as follows. Section 2 recalls the main features of Ciaramella and its compiler, Zampogna. Section 3 describes the new Ciaramella web playground.

2. CIARAMELLA AND ZAMPOGNA

Ciaramella is an audio specific programming language aimed at describing DSP systems. Its declarative nature and SDF-based model result in a light and unconstrained syntax which guarantees an high-level of modularity and flexibility. Even more interestingly, it permits the description of delay-free feedbacks between subsystems. This makes it possible for known models that are hard to code such as wave digital filters (WDF) to be programmed easily [7].

As simplicity and minimalism are Ciaramella's design goals, a small set of programming abstractions is provided which reflect the SDF concepts and make the description of DSP systems possible. In particular, the fundamental components are:

- the **block**, that encapsulates an operation and it may be simple (atomic) or composed by other blocks (composite);
- input or output **ports**, attached to a block and representing connection endpoints;
- the **connection**, that defines a directed flow of data (e.g. a signal) between two ports.

[1] https://faustide.grame.fr/
[2] https://kronoslang.io/veneer/
[3] https://soul.dev/lab/
[4] https://emscripten.org/

The language supports some built-in atomic blocks for the basic arithmetic operations and for the delay operation, i.e. the **unit delay** block, which is necessary to produce computable loops.

2.1 Syntax

Since Ciaramella is designed to cope with high-level manipulation of data streams, every variable or expression refers to entire flows rather than instantaneous values. For example the

```
a = b + c
```

code specifies that, for every temporal sample n, $a_n = b_n + c_n$.

A Ciaramella program is made of a list of statements, which can be assignments or composite block definitions. An assignment is in the simple form:

```
id1, id2 = expression
```

and it also acts as a declaration of the `id1` and `id2` variables. Indeed, in Ciaramella a variable can be declared and assigned only once and the order of the statements is not relevant.

The syntax for composite block definition is:

```
y1, y2 = composite_block_id (x1, x2, vol) {
        tmp = x1 + x2
        y1 = y2 * vol
        y2 = tmp / 2
}
```

It defines a reusable block (like a function), referable as `composite_block_id`, which has 3 input ports (`x1`, `x2` and `vol`) and 2 output ones (`y1`, `y2`). The body contains the assignments and expressions for calculating the values of the output. The output ports must be always assigned and can be used as normal variables, while the input ones are considered external and cannot be re-assigned.

A composite block can be used following a C-like function call syntax. This operation is called "composite block instantiation". For example:

```
t1, t2 = composite_block_id (in1, in2, 0.3)
```

instantiates a `composite_block_id` and redirects its outputs to the newly created `t1` and `t2` variables.

The built-in unitary delay block is of particular interest for the creation of computable feedbacks. The following example implements an iteration counter and shows its syntax:

```
counter = delay1(counter) + 1
@counter = 0
```

The `delay1(counter)` expression returns the value of `counter` at the $n-1$th iteration, while the special `@` symbol sets the initial value of `counter`, which is needed for the first iteration.

A typical Ciaramella program consists in a list of composite block definitions and assignments of constants. Each composite block has its local scope, while constants are globally scoped and can be accessed anywhere within the code.

The following example is the implementation of three composite blocks representing some trivial low-pass filtering functions.

Listing 1. Low-pass filters in Ciaramella

```
b = 0.1
y = lp (x) {
        y_z1 = delay1(y)
        y = y_z1 + b * (x - y_z1)
        @y = 0
}
y = lp3 (x) {
        y = lp(lp(lp(x)))
}

yL, yR = lp3_stereo (xL, xR, volume) {
        yL = lp3(xL) * volume
        yR = lp3(xR) * volume
}
```

For a more advanced example that implements a WDF based low-pass filter, check [7].

2.2 Compiler

We implemented a compiler for Ciaramella called Zampogna, written in JavaScript. Its entire codebase consists of about 1500 lines of code. Zampogna is able to produce C++ with VST2 wrapper, MATLAB, and lately JavaScript with Web Audio wrapper.

The compilation process consists of several steps, from the Ciaramella code parsing to the target code generation. A distinctive step is the production of an intermediate graph representation (IG) starting from the abstract syntax tree (AST). The IG reflects the SDF process network which in turn corresponds to the DSP system described by the Ciaramella code. The IG gets flattened, in order to reduce the system to atomic blocks, then optimized, and finally scheduled for sequential execution. The SDF formalism allows the scheduling to be accomplished statically (at compile time) [8], which is fundamental to maintain high performances of the DSP algorithms. The last phase of the compiler is the production of target code. To accomplish this, it makes use of the doT templating library [9], which enhances modularity allowing to easily add support for further target languages.

Zampogna can be used via the `zampogna-cli.js` command line interface. For instance, to compile the previous low-pass filters example (listing 1), assuming the code is saved in the `lp.crm` file, the following command can be used:

```
zampogna-cli.js -i lp3_stereo -t cpp
-c volume lp.crm
```

The `-i lp3_stereo` option specifies that `lp3_stereo` composite block acts as *initial block*, that is like the typical C *main* function. The `-t cpp` option selects C++ as target language and `-c volume` communicates to the compiler that the `volume` input is an user input control; the other inputs are assumed to be normal audio data flows (audio rate).

We highlight that writing the compiler purely in JavaScript with almost no external dependency, makes it natively embeddable in any web page.

3. WEB IDE/PLAYGROUND

We developed a fully functional Ciaramella web playground. In order to accomplish it, we first added JavaScript as target language to Zampogna. Then, we generated the web compatibile version of Zampogna, `zampogna-web.js`, through *browserify* [5]. A `zampogna-cli.js` is also available as command line interface to be used via Node.js [6]. Finally, we created the HTML web page that acts as an all-in-one text editor, compiler and execution environment.

Both the language and the web playground are at an experimental stage. Zampogna source code is available at https://github.com/paolomarrone/Zampogna, while the web playground can be tested at https://ciaramella.dev/.

3.1 User Interface

The code text editing section features tabbing to allow working on more projects simultaneously. Alongside the text editor, input areas for the necessary compiler options are present (mainly: initial block and control inputs information). A read-only console box has been placed below the "Compile" button to retrieve the compilation messages from Zampogna. This section is showed in Figure 1.

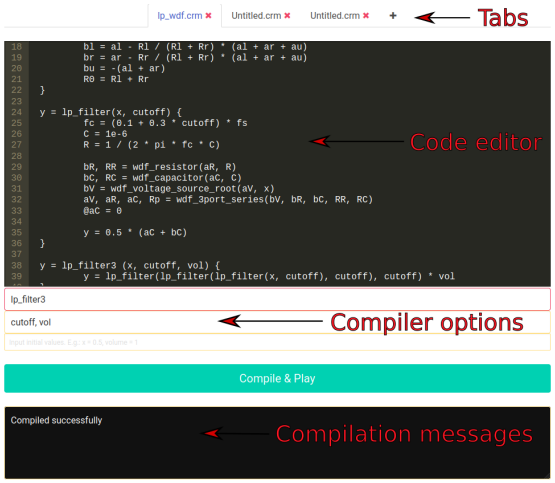

Figure 1. Ciaramella web editor

In case of valid input and successful compilation, the produced JavaScript code serves as input for the execution section. This section consists in a small Web Audio nodes network composed by an audio source node, the custom node that incorporates the output of the compilation, and an output node that represents the environment audio system. As source node, it is possible to choose between the system microphone (*MediaStreamAudioSourceNode*), or a file from both server or client side (*AudioBufferSourceNode*). The custom node is, instead, an *AudioWorkletNode*.

[5] `https://browserify.org/`
[6] `https://nodejs.org/en/`

The execution section dynamically generates the graphical elements for the manipulation of the user controls starting from the compilation options specified by the user. They are simple HTML input elements of range type. Figure 2 shows an example of this section.

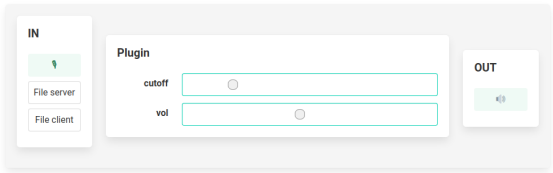

Figure 2. Ciaramella web player

The last section of the page is devoted to the plugin export feature (Figure 3), allowing to build plugins in C++, MATLAB, and JavaScript language. It is possible, then, to view and download them.

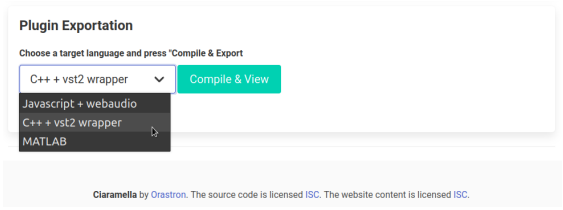

Figure 3. Ciaramella web plugin exportation

We did use the Bulma CSS library [7] for the styling. The whole webpage, including custom styling and scripts, amounts to about 400 lines of code.

3.2 Implementation Details

The output of the compilation for the execution section is a JavaScript program in form of string (`procJsStr`). It contains the code of the audio processing algorithm and an *AudioWorkletProcessor* wrapper.

Typically, `procJsStr` would be saved in a file (commonly `processor.js`) for later inclusion as a module to the audioWorklet. This procedure is not viable in our case because `procJsStr` is generated dynamically at client-side and we want the server that hosts the webpage to be completely unaware of what the user does. Our solution relies on the Blob JavaScript object [8], that is more abstract than the File one. Indeed, the File interface is based on Blob. If `ctx` is the *AudioContext* and the custom *AudioWorkletProcessor* is registered as `PluginProc`, the code to create the custom node is:

```
let scriptUrl = URL.createObjectURL(new Blob(
[procJsStr], {type: "text/javascript"}));

await ctx.audioWorklet.addModule(scriptUrl);

let customNode = new AudioWorkletNode(ctx,
"PluginProc", { outputChannelCount: [1] });
```

[7] `https://bulma.io/`
[8] `https://developer.mozilla.org/en-US/docs/Web/API/Blob`

In a realistic scenario, the compilation is repeated many times, and the "module substitution" problem arises: the *AudioWorklet* provides the *addModule()* method but no method for deleting or editing of the existing ones. Consequently, modules accumulate over and over and, for long sessions, they may encumber the memory. There are at least two ways to handle this problem.

The first one is the destruction and recreation of the whole *AudioContext* object: this is the simplest and drastic solution.

In the second solution, instead, there exist only one context and one module, but the latter has to provide a mechanism to change its inner audio processing part. In particular, the output of the compilation needs to be sent via the *AudioWorkletNode* port messaging system to the *AudioWorkletProcessor*, which then uses the built-in JavaScript *eval()* function [9] to substitute and evaluate its own inner code. This solution, unfortunately, has some efficiency problems because *eval* constrains the JavaScript interpreters or engines from performing ahead of time optimizations [10]. More specifically, modern browsers tend to convert JavaScript scripts to machine code, losing variable naming informations; consequently the not-optimized code passed to *eval* causes long and expensive lookups within the machine code. Moreover, it leads to security risks based on the ease for bad actors to execute arbitrary code.

Ultimately, since *eval* is generally discouraged [10] and since we require the code to be as fast as possible, we chose to implement the first solution.

4. CONCLUSIONS

We developed a fully functional web playground for Ciaramella. It is now possible for both domain experts and newcomers to try out Ciaramella on the fly without being limited by installation and cumbersome testing processes. We showed how JavaScript was a convenient choice for the compiler code: we obtained both small codebase size and native execution on the web. Furthermore, this work is a platform for future development: the next target language we aim to support is WebAssembly, which will be used for the audio processing part of the web audio plugins. It will grant near-native performances and make plugins usable in real applications rather than for prototyping purposes.

5. REFERENCES

[1] E. A. Lee and D. G. Messerschmitt, "Synchronous data flow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, 1987.

[2] Y. Orlarey, D. Fober, and S. Letz, "Syntactical and semantical aspects of FAUST," *Soft Computing*, vol. 8, no. 9, pp. 623–632, 2004.

[3] V. Norilo, "Kronos: a declarative metaprogramming language for digital signal processing," *Computer Music Journal*, vol. 39, no. 4, pp. 30–48, 2015.

[4] "Max/gen." `https://docs.cycling74.com/max8/vignettes/gen_topic`. Accessed: 2022-09-14.

[5] "Web audio API." `https://www.w3.org/TR/webaudio/`. Accessed: 2022-05-27.

[6] S. Letz, Y. Orlarey, and D. Fober, "Compiling faust audio DSP code to webassembly," in *Web Audio Conference*, 2017.

[7] P. Marrone, S. D'Angelo, F. Fontana, G. Costagliola, and G. Puppis, "Ciaramella: a synchronous data flow programming language for audio DSP," in *Sound and Music Computing Conference*, SMC, 2022.

[8] E. A. Lee and D. G. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE Transactions on computers*, vol. 100, no. 1, pp. 24–35, 1987.

[9] "dot - the fastest + concise javascript template engine for node.js and browsers.." `https://olado.github.io/`. Accessed: 2022-05-27.

[10] S. H. Jensen, P. A. Jonsson, and A. Møller, "Remedying the eval that men do," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pp. 34–44, 2012.

[9] `https://tc39.es/ecma262/multipage/global-object.html#sec-eval-x`

[10] `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/eval`