From Diagram to Code: a Web-based Interactive Graph Editor for Faust DSP Design and Code Generation

Shihong Ren, Laurent Pottier, Michel Buffa

Proceedings of the 2nd International Faust Conference (IFC-20), Maison des Sciences de l'Homme Paris Nord, Saint-Denis, France, December 1-2, 2020

FROM DIAGRAM TO CODE: A WEB-BASED INTERACTIVE GRAPH EDITOR FOR FAUST DSP DESIGN AND CODE GENERATION

Shihong Ren

Université Jean Monnet
Saint-Etienne, France
renshihong@hotmail.com

Laurent Pottier

Université Jean Monnet
Saint-Etienne, France
laurent.pottier@univ-st-etienne.fr

Michel Buffa

Université Côte d'Azur, CNRS, INRIA
Sophia Antipolis, France
michel.buffa@univ-cotedazur.fr

ABSTRACT

FAUST, as a domain-specific language (DSL) initially designed to represent block-diagram algebra [1], is efficient through its syntax to describe DSPs because of its structural similarity to physical signal processing devices. This approach allows the FAUST compiler to generate an equivalent graphical representation of the coded algorithm as a block-diagram. Considering the nature of the language, the reverse should be possible as well (compiling an equivalent FAUST DSP code from a block-diagram). This paper discusses the possibility to achieve this process and proposes a web-based block-diagram graph editor for designing FAUST DSP.

1. INTRODUCTION

Many visual programming languages (VPLs) such as Max [2] or PureData [3] provide a graphic canvas to allow developers to connect functions or data between them. This canvas with connected objects, also known as a patcher [4], is basically a graph, meant to be interpreted as dataflow computation [5] by the system. These visual languages are user-friendly as they seem closer to the way things work in our physical world, especially in the audio processing field. Connecting signal processors using audio cables to produce sounds and effects is a common practice even though we can now bring this practice to the digital world. This is one of the reasons for the design of the FAUST programming language, that represents its code with its patcher-like graph called *block-diagram algebra* (BDA) [6].

By adopting the BDA as the core of the language, the drawbacks of the graph representation and dataflow model [7] are discussed mainly due to the complexity and difficulty on the performance optimization aspect and the lack of an explicit semantic of the algorithm written. As a result, FAUST is designed to be closer to machine-friendliness. It is, in the end, a text-based language that can be represented as a BDA that is optimized and transformed into a high-performance low-level code.

So, even if the BDA is "flattened" internally, the FAUST code is always represented by a graph that leaves the possibility to generate code from an equivalent graph. To understand or design a DSP, a patcher or a block-diagram is often more illustrative for audio developers than mere source code, the visual representation looks like electric/electronic diagrams and hides all the "plumbery," the developer does not have to master the details of the implementation language. In addition, some VPLs can group subgraphs into blocks and provide zoomable views of the diagram logic, making it easier to find errors. This is why we think that an editor for such diagrams would be a desirable addition/improvement to an Integrated Development Environment (IDE) for FAUST [8]. A graph-to-code approach would help writing DSP algorithms, boosting the design while not impacting the performance.

Since Max and PureData, patcher-like VPLs are massively developed especially on the web. WebPd[1] is a web-based PureData patcher interpreter using JavaScript and the WebAudio[2] API. Cables.gl[3] is a video-oriented patcher editor on the web that also handles WebAudio nodes. WebAudio Visual Editor,[4] WebAudioDesigner,[5] Mosaicode[6] [9] and Olos[7] are web-oriented VPLs for audio processing. Still, using FAUST as the graph-to-code compilation target will bring as well the advantage of being able to export efficient and robust DSPs to any supported platform including Max, PureData and WebAudio. Along with the development of the FAUST compiler, the presented work may be considered as a universal DSP design tool for many DSP platforms.

In the next sections, we will take a look at Gen,[8] one of the most popular existing graph-to-code generators for DSP, then discuss the possible implementations of FAUST's different features in a graph editor. Finally, we will propose a web-based application that implements the first version of the editor and the code generator.[9]

2. GEN IN MAX

As a module of Max visual programming language, Gen adopts the same patcher graph representation as its main graphical user interface (GUI). But instead of computing dataflow based on the graph internally, Gen pre-compiles the graph into a text-based intermediate code using the language GenExpr.[10] Before runtime,

[1] https://github.com/sebpiq/WebPd
[2] https://www.w3.org/TR/webaudio
[3] https://cables.gl/
[4] https://github.com/pckerneis/WebAudio-Visual-Editor
[5] https://github.com/g200kg/webaudiodesigner
[6] https://mosaicode.github.io/
[7] https://www.jasonsigal.cc/portfolio/olos.
[8] https://docs.cycling74.com/max8/vignettes/gen_overview
[9] https://fr0stbyter.github.io/jspatcher/dist/?mode=faust
[10] https://docs.cycling74.com/max8/vignettes/gen_genexpr

the GenExpr code is interpreted to a lower-level code to get higher performance.

The GenExpr code that the system compiles to is available to users who can also write directly GenExpr code as a part of the program. This makes the language rather similar to FAUST and serves as inspiration. In the presented work, we follow certain paradigms from Gen to offer developers a consistent experience across platforms.

2.1. Inlets, Outlets, Inputs and Outputs

When a Gen patcher is compiled, the boxes and lines in the patcher are analyzed to generate the GenExpr code. The boxes with different text on them represent different functions. They have a certain number of inlets and outlets as their function parameters and results (inputs and outputs). The text on the box can be followed by arguments to override its parameters and to suppress the inlets (Figure 1). Unconnected inlets will be replaced by 0 while compiled.

Figure 1 *Gen example: the "+" function has two inlets by default, one will be suppressed by adding an argument*

When a DSP diagram is built by a Gen patcher, it should have at least one output. Outputs are the starting points of the analysis. Inputs are involved in the code generation only if a path from input to output can be found. Inputs and outputs are represented as boxes named `in` and `out`. For example, Figure 2 will be compiled to:

```
out1 = in1 + in2;
out2 = in2;
out3 = 0;
```

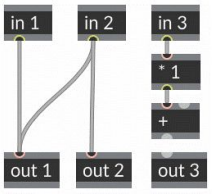

Figure 2 *Gen example: inputs and outputs*

Figure 2 also shows that when two lines merge into one inlet, they will be added together. When two lines are coming from the same outlet, they will represent the same signal and be named identically in the code.

2.2. Subprocess and Identifiers

A Gen patcher can contain multiple Gen patchers or GenExpr code blocks. They can be considered as reusable user-defined functions. This approach is similar to defining a reusable function in Faust. A sub-patcher in Gen will have the same number of inlets and outlets as the amount of `in` and `out` objects used; the code block is also connectable when the code is properly written and compiled. For example, the patcher in Figure 3 has one code block and one sub-patcher that represent the same function. So, the resulting code is:

```
expr_1 = max(in1, in2);
out1 = expr_1;
max_2 = max(in1, in2);
gen_3 = max_2;
out2 = gen_3;
```

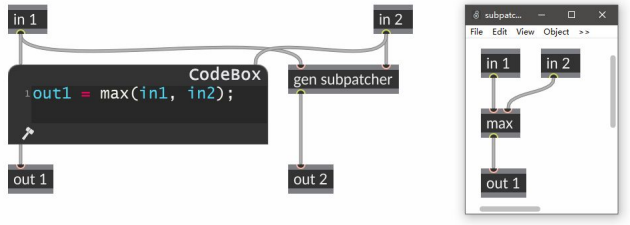

Figure 3 *Gen example: sub-patcher and code block*

In this example, we also understand that each result of a function is named with an identifier formed by the concatenation of the name of the function or outlet, an underscore, and a unique number. These identifiers are thereby unique and will be reused if the result is passed to other functions. Although the system can add some redundancy, it allows the developer to easily find out the relation between the graph and the code.

2.3. Loops

Dealing with loops is a common and critical issue in DSP language design. In Gen, it is possible to create a loop only if a delay of at least one sample appears in the loop. The graph is similar to the block-diagram that a FAUST recursive composition produces, which creates automatically a one-sample delay. Figure 4 is an example of simple feedback with a one-sample delay function `history`.

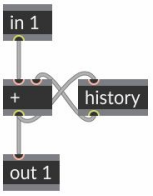

Figure 4 *Gen example: a feedback with one-sample delay*

The graph is compiled to the following code:

```
History history_1(0);
add_2 = in1 + history_1;
out1 = add_2;
history_1_next_3 = fixdenorm(add_2);
history_1 = history_1_next_3;
```

The loop with the `history` object generates the code differently, as the delay is declared explicitly on the top, with an extra `fixdenorm` function to replace the denormal values by 0.

2.4. Explicit and Implicit

FAUST is a functional language that can be used to build concisely a graph. This allows us to create new functions by chaining defined functions. However, once a new function is written, it

becomes difficult to tell its exact number of inputs and outputs until the code is compiled. The same implication happens when we compose blocks using operators: it is not possible to tell before compilation how each input and output are connected without deep information of blocks.

We chose to adopt in the work presented in this paper Gen's graph representation which is explicit in terms of I/Os and connections, as one inlet or outlet represents exactly one signal. This, on one hand, may raise some limitations when one needs to connect a large amount of I/Os or need to create a function with a variable number of arguments (like *pattern-matching* in FAUST [10]), but, on the other hand, brings an easily understandable graph that matches the block-diagram being compiled.

3. COMPILE A GRAPH TO FAUST

3.1. Functions

Basically, in FAUST, each block (or box in the graph) can be transformed into a `f(x)` expression (with some exceptions) [11]. The expression (prefix application syntax) is widely used in FAUST even for algebra symbols. As an example, `a - b` is equivalent to `-(a, b)`. This feature allows us to generalize the way a functional box can be transformed into FAUST code.
Table 1 shows how we generate expressions from an unconnected, no-argument box with different numbers of I/Os.

Table 1 *Code generated from unconnected functional boxes*

text	code
`+`	`Add_1_0 = +(0, 0);`
`sin`	`Sin_1_0 = sin(0);`
`ma.SR`[11]	`ma_SR_1_0 = ma.SR;`
`en.ar`[12]	`en_ar_1_0 = en.ar(0, 0, 0);`

`os.oscrq`[13]	`os_oscrq_1 = os.oscrq(0);`
	`os_oscrq_1_0 = os_oscrq_1 : _, !;`
	`os_oscrq_1_1 = os_oscrq_1 : !, _;`

The generation rules are:

1. Inlets and outlets are filled by 0 if not connected.
2. If the number of inlets is 0, which means the function has no input, then no parameter will appear in the code.
3. The outlets are named from the function name, box's unique identifier number and the outlet index.
4. If the function has multiple outputs, an intermediate expression will be generated before splitting to each outlet.

In Gen, the arguments attached after the function name in the box will be used to replace some function inputs by constant values. To achieve this, we added more generation rules:

5. The arguments defined will be completely or partially applied to the beginning of available function inputs except for infix operators and their relatives.[14] For these exceptions, the arguments will be applied at the end of the available function inputs.
6. The _ (also known as *identity block*) is used in FAUST to bypass a signal. We consider it as a placeholder in arguments to keep the inlet available.

Examples of unconnected boxes with arguments (Table 2):

Table 2 *Code generated from unconnected functional boxes with arguments*

text	code
`+ 1`	`Add_1_0 = +(0, 1);`
`@ 10`	`Delay_1_0 = @(0, 10);`
`en.ar _ 0.1`	`en_ar_1_0 = en.ar(0, 0.1, 0);`

Now more rules about inlets: (Figure 5)

7. If an inlet is connected by one line, the identifier of the related outlet will be assigned to the function input.
8. If connected by more than one line, the outlets related will be merged into one signal before being assigned to the function input.
9. Inlets are one by one assigned to function inputs from start, skipping which is already replaced by arguments.

Table 3 *Code generated from connected functional boxes*

text	code
`+`	`Add_3_0 = +(In_1_0, In_2_0);`
`/ 1 _`	`Div_3_0 = /(1, (In_1_0, In_2_0 :> _));`
`en.ar _ 0.1`	`en_ar_3_0 = en.ar(In_1_0, 0.1, In_2_0);`

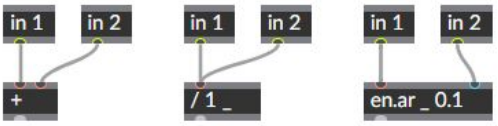

Figure 5 *Graphs of Table 3*

For pattern-matching functions, the number of I/Os is variable. To make these functions usable in the graph, we allow setting the number of function I/Os explicitly using Gen's attribute syntax. For example, `si.bus 5 @ins 6 @outs 5` declares that the function has 6 inputs and 5 outputs, then creates 5 inlets for the box as it already has one argument.

3.2. Main Process and Loops

The graph representation for loops in FAUST's block-diagram or Gen is straightforward and intuitive. But compiling the graph to FAUST code is problematic. Indeed, if the loop is on the same level as other functions, the graph of the loop (recursion) is determined by the available number of I/Os of the process within the loop. In code generation, the recursive composition sign cannot be correctly placed unless the subprocess is compiled without any other loops. So, it is better to analyze these recursions with a higher priority and place them on an upper

[11] `ma.SR` has 0 input, 1 output.
[12] `en.ar` has 3 inputs, 1 output.
[13] `os.oscrq` has 1 input, 2 outputs.
[14] Functions that arguments are applied to the end of inputs in our implementation (the list is slightly different from Faust's infix operators): `+, -, *, /, ^, %, &, |, xor, @, <, <=, >, >=, <<, >>, ==, !=, pow, atan2, fmod, remainder, rdtable, rwtable.`

level. As a result, these recursions are set aside and our main process will have a part of its I/Os dedicated to them.

For example, a process that has no input, 3 outputs and contains one recursion will be restructured as following (Figure 6):

```
Main(Rec_4_0) = Rec_4, Out_1, Out_2, Out_3 with {
    Rec_4 = ...;
    Out_1 = ...;
    Out_2 = ...;
    Out_3 = ...;
};
Rec = _ : _;
process = Main ~ Rec : !, _, _, _;
```

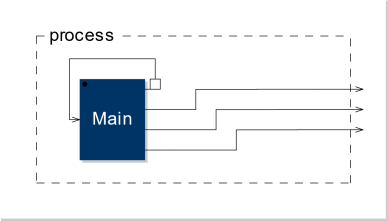

Figure 6 *Restructuring a process with recursion*

We are now able to compile the following graph (Figure 7):

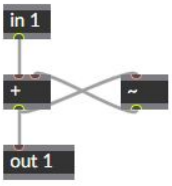

Figure 7 *A simple feedback to compile*

To the code:

```
Main(Rec_3_0, In_2_0) = Rec_3, Out_1 with {
    Add_4_0 = +(In_2_0, Rec_3_0);
    Rec_3 = Add_4_0;
    Out_1 = Add_4_0;
};
Rec = _ : _;
process = Main ~ Rec : !, _;
```

3.3. Subprocess

Three subprocess design patterns are proposed: code in the graph, sub-patcher, FAUST iterations.

3.3.1. Code Block

FAUST offers several λ-function style syntaxes to facilitate the encapsulation of a subprocess. To integrate a separate FAUST DSP into a new process, we need the `environment{...}` syntax to ensure its independence and use its main process as our subprocess. This approach causes fewer problems as the FAUST compiler only allows using `import` syntax in a new environment.

The example below (Figure 8) shows how to integrate an independent FAUST code into the graph. In this example, the patcher already evaluated the code block to create two inlets and one outlet.

Figure 8 *FAUST code block in graph*

This graph will be compiled to the following code:

```
process(In_2_0, In_3_0) = Out_4 with {
    Code_1_0 = environment{
        process = max;
    }.process(In_2_0, In_3_0);
    Out_4 = Code_1_0;
};
```

3.3.2. Sub-patcher

A patcher with outputs can be included in another patcher as a subprocess. We use FAUST's `with {...}` expression which creates a local scope.

If we put the patcher of Figure 8 in a new patcher (Figure 9):

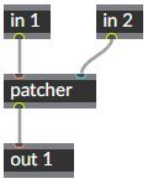

Figure 9 *Figure 8 as a sub-patcher*

The resulting code will be:

```
process(In_2_0, In_3_0) = Out_4 with {
    SubPatcher_1_0 = process with {
        process(In_2_0, In_3_0) = Out_4 with {
            Code_1_0 = environment{
                process = max;
            }.process(In_2_0, In_3_0);
            Out_4 = Code_1_0;
        };
    };
    Out_4 = SubPatcher_1_0;
};
```

3.3.3. Iterations

FAUST offers four iteration functions: `par`, `sum`, `prod` and `seq`. Compared to other functions, they work with a variable that is local to their scope, which needs to be accessible in a local graph as a subprocess.

We propose a loop-like graph, using the same box to provide this scoped variable and to retrieve the sub-process graph. The iteration box will take from its inlet the sub-graph to repeat.

This is an example of a parallel iteration which has 3 outputs (Figure 10), the fourth outlet is the local variable. The `par` box will look for a graph above its first inlet to take as its sub-process.

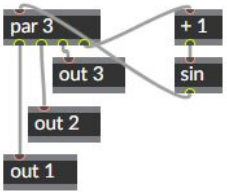

Figure 10 *`par` iteration graph example*

```
process = Out_1, Out_2, Out_3 with {
    Par_4 = par(Par_4_3, 3, lambda with {
        lambda = Par_4 with {
            Add_5_0 = +(Par_4_3, 1);
            Sin_6_0 = sin(Add_5_0);
            Par_4 = Sin_6_0;
        };
    });
    Par_4_0 = Par_4 : _, !, !;
    Par_4_1 = Par_4 : !, _, !;
    Par_4_2 = Par_4 : !, !, _;
    Out_1 = Par_4_0;
    Out_2 = Par_4_1;
    Out_3 = Par_4_2;
};
```

3.4. UI Components

As FAUST offers some primitives for UI description (checkbox, button, sliders, etc.), we implement them in the graph. Here is an example (Figure 11):

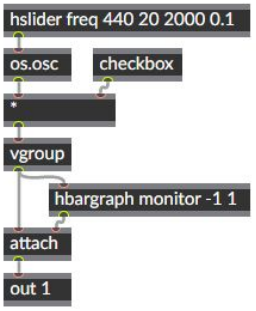

Figure 11 *Graph with UI descriptors*

The graph is similar to a FAUST block-diagram. The UI component name is optional. If omitted, it will be named using its box identifier.

```
import("stdfaust.lib");
process = Out_1 with {
    HBargraph_2_0
    = VGroup_3_0 : hbargraph("monitor", -1, 1);
    Checkbox_7_0 = checkbox("Checkbox_7");
    HSlider_8_0
    = hslider("freq", 440, 20, 2000, 0.1);
    os_osc_6_0 = os.osc(HSlider_8_0);
    Mul_5_0 = *(os_osc_6_0, Checkbox_7_0);
    VGroup_3_0 = vgroup("VGroup_3", Mul_5_0);
    Attach_4_0
    = attach(VGroup_3_0, HBargraph_2_0);
    Out_1 = Attach_4_0;
};
```

4. WEB-BASED GRAPH EDITOR

The FAUST compiler is available altogether with an IDE on the Web platform since 2015 [12, 13]. Recently, W3C Web standards such as WebAudio, WebMIDI,[15] WebAssembly,[16] opens a considerable potential for the FAUST ecosystem. For example. a ready-to-use JavaScript module named `faust2webaudio`[17] has been developed in 2019, which allows us to evaluate FAUST DSP code and to compile to WebAssembly/WebAudio code that can run directly in a browser [14]. Web and web browsers, thanks to their accessibility and their increasing performance, became an attractive environment to develop FAUST-related tools.

Consequently, a modern web development tool-chain has been used to develop our editor, to ensure the efficiency of coding and compatibility across browsers.

4.1. Tool-chain

We are using TypeScript[18] as our main development language which provides an additional level of maintainability compared to JavaScript. It also supports multiple tools that we are working with: React,[19] a framework we used mainly for the UI layer; Babel,[20] a set of JavaScript compilation utilities to keep our code runnable on different browser versions; Webpack,[21] a production tool that manages code dependencies, etc.

For the UI components based on React, we are using the React version of Semantic UI[22] for the overall look and feel; the Monaco[23] source code editor, SCSS[24] for page layout design, etc.

4.2. Data Structure

Inspired by Max and Gen's patcher structure, a patcher is basically an augmented graph with lines and boxes. They can be described in the form "Map" data structures: sets of key-value pairs (or more precisely key-objects). Their key/identifiers (IDs) are generated automatically using auto-incrementing numbers so that lines and boxes can be arranged by their creation order.

```
interface Patcher {
    lines: Record<string, Line>;
    boxes: Record<string, Box>;
}
```

[15] https://www.w3.org/TR/webmidi
[16] https://webassembly.org/
[17] https://github.com/grame-cncm/faust2webaudio
[18] https://www.typescriptlang.org/
[19] https://reactjs.org/
[20] https://babeljs.io/
[21] https://webpack.js.org/
[22] https://react.semantic-ui.com/
[23] https://microsoft.github.io/monaco-editor/
[24] https://sass-lang.com/

Lines have some properties, such as their source and destination box. Boxes hold more information: number of I/Os, text content, position, dimensions, behavior, etc.

```
interface Line {
    // source box ID and port index
    src: [string, number];
    // destination box ID and port index
    dest: [string, number];

}
interface Box {
    inlets: number;
    outlets: number;
    text: string;
    // Position and dimensions
    // [left, top, width, height]
    rect: [number, number, number, number];
    // Behavior
    object: FaustOp;
}
```

The `FaustOp` object controls the lifecycle of the box: determines what to do when the box is created, connected, disconnected, destroyed, etc. It is also designed to output a part of FAUST code when requested. We will discuss this aspect in further subsections.

Patchers, lines, and boxes are all event emitters that fire different events when things happen. This mechanism allows us to separate them from the UI system in terms of data structure (loose coupling). UI components, since their creation, subscribe to these events to properly react to data changes. For example, each time the graph changes (is being edited), a new FAUST code will be generated (Figure 12).

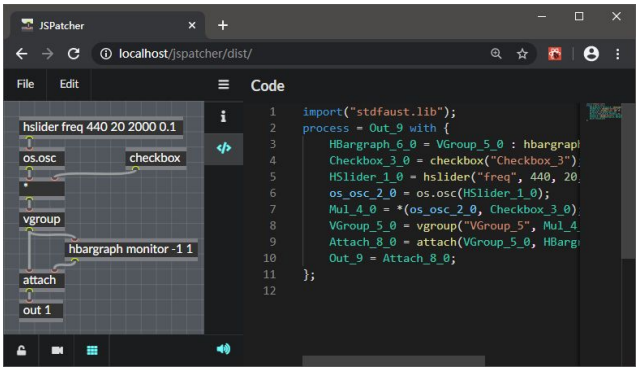

Figure 12 *The generated code can be previewed in real-time on the right panel (synchronized to the graph)*

4.3. FAUST Standard Library Analysis

The `faust2webaudio` module which is included in the web version of the FAUST compiler comes with the FAUST's standard library `stdfaust.lib`. [15] To use the functions from the library, we need to get a full list of available functions.

All these library functions are well documented with a shared common format. To get a full list of available functions, a single traversal with a parser returns all the function names, and this result can be used both for syntax highlighting and for auto-completion in the code editor (also known as "IntelliSense features").

As the I/Os of these functions are implicit, the patcher needs to evaluate a function with the FAUST compiler to get their number of I/Os and to create box inlets and outlets. This is achieved by using a predefined method of `faust2webaudio`.

4.4. Graph Analysis

Code generation is the main task of the presented system. It regenerates the code by analyzing the graph in real time. The analysis is performed from bottom to top, more precisely, from each `out` box as the root of the tree traversal.

This first full traversal is meant to find out all the boxes that are connected in the graph and to name their outlets using a unique FAUST identifier. A map is then created, that holds the relationship between lines and identifiers of their source outlet. Then a second traversal is performed: we call a method on all the boxes with the map as an argument, this method returns for each box the corresponding expression. These expressions are concatenated to the final code without any particular order. Indeed, most of the time, the FAUST compiler does not care about the order of expressions. For rare ones it does care, in an extra step, we locate them to the top of the code.

As an example, here (Figure 13) is a graph with one output.

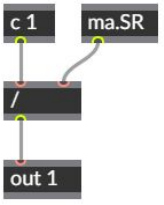

Figure 13 *Code generation steps example*

Our first traversal brings back a map and visited boxes. The boxes are (Table 4):

Table 4 *Boxes visited*

box ID	box text
1	`out 1`
2	`/`
3	`ma.SR`
4	`c 1`[25]

The map looks like that (Table 5):

Table 5 *Line map*

line ID	FAUST code identifier
`line-1`	`Div_2_0`
`line-2`	`ma_SR_3_0`
`line-3`	`Const_4_0`

Then we get the expressions from the boxes. They are (Table 6):

[25] `c` represents a constant value.

Table 6 *Expressions generated*

box text	expressions
`out 1`	`Out_1 = Div_2_0;`
`/`	`Div_2_0 = /(Const_4_0, ma_SR_3_0);`
`ma.SR`	`import("stdfaust.lib");` `ma_SR_3_0 = ma.SR;`
`c 1`	`Const_4_0 = 1;`

As `ma.SR` generated a special expression `import(...);` that should be included only once in the final code, we put it at the top.

Then we wrap the rest together with the main process using the `out`'s identifier:

```
import("stdfaust.lib");
process = Out_1 with {
    ma_SR_3_0 = ma.SR;
    Const_4_0 = 1;
    Div_2_0 = /(Const_4_0, ma_SR_3_0);
    Out_1 = Div_2_0;
};
```

5. Compile a Gen Patcher to Faust

Since we are using the Gen approach to design the Faust graph-to-code compiler, it is, therefore, possible to parse a Gen patcher and compile it to the Faust code.

To achieve this process, we implemented all the Gen operators with Faust functions in a separate Faust library file `gen2faust.lib` which will be imported as a dependency of the generated Faust code. Using this library file, we can interpret Gen boxes as Faust expressions according to their box texts.

There are some differences in the interpretation of the box texts compared to the Faust ones:

1. When Gen boxes are compiled to GenExpr, unconnected inputs can have their default value instead of 0.
2. Attributes (additional properties) can change the box's behavior.

For example, the Gen `cycle` box has one input and two modes defined by the attribute `@index`. If the mode is set to `freq`, its input is 440 by default. We prepared two Faust functions `cycle` and `cycle_phase` for these two modes (Table 7).

Table 7 *Compile Gen boxes*

box text	expressions
`cycle @index freq`	`cycle(440);`
`cycle @index phase`	`cycle_phase(0);`

By implementing these features to the editor, we managed to transform some Gen patchers to the Faust code. However, some Gen operators which manipulate Max buffers remain difficult to interpret as Faust's audio import is not supported yet in the WebAssembly version of the Faust compiler.

6. CONCLUSIONS

The traditional workflow, when developing DSP code with Faust, is firstly to have a graph in mind, then transform it into code, then check if the code matches the original graph by comparing it visually with the diagram output by the compiler. The graph-to-code system presented in this paper sort of reverses this workflow: design graphically the graphic DSP algorithm, then generate the Faust code. No need to check if the code matches the graph anymore. In addition, for pedagogical purposes, it can help beginners to easily build their DSP from scratch and to understand how Faust works.

Furthermore, thanks to modern web technology such as WebAssembly and the WebAudio API, we are able to compile, debug and run Faust DSP just-in-time in a web browser. This can transform our system into another more graphical oriented, online IDE-like tool. For the moment, there are still some features in Faust that are not yet implemented in the project; but, eventually, using such a patcher system, we will be able to manipulate between multiple Faust DSPs or even with other WebAudio-compatible DSPs in a single workspace.

The graph editor with the code generator is being developed as an open-source project and is functional online with a video demonstration.[26] It can save and load patchers as a JSON file. Thanks to `faust2webaudio` and loose decoupling, it is possible to compile a graph file to Faust code or to a WebAudio Faust DSP automatically without any UI involved. So, the system can be integrated into Faust IDEs or any WebAudio applications such as the WASABI Pedalboard.[27] We are also developing a web-based VPL for music composition and performance that use this project as its module.

7. References

[1] Y. Orlarey, D. Fober and S. Letz, "Syntactical and Semantical Aspects of Faust," *Soft Computing,* 2004.

[2] M. Puckette and D. Zicarelli, "Max/msp," *Cycling,* 1990.

[3] M. Puckette, "Pure Data," in *Proceedings of the International Computer Music Conference*, Thessaloniki, 1997.

[4] M. Puckette, "The patcher," in *Proceedings of the International Computer Music Conference*, San Francisco, 1986.

[5] E. A. Lee and T. M. Parks, "Dataflow process networks," *Proceedings of the IEEE,* vol. 83, pp. 773-801, 1995.

[6] Y. Orlarey, D. Fober and S. Letz, "An algebra for block diagram languages," in *Proceedings of the International Computer Music Conference*, Gothenburg, 2002.

[7] R. Stephens, "A survey of stream processing," *Acta Informatica,* vol. 34, pp. 491-541, 1997.

[8] R. Michon and Y. Orlarey, "The Faust Online Compiler: a Web-Based IDE for the Faust Programming Language," in *Proceedings of the Linux Audio Conference*, Stanford, 2012.

[26] https://youtu.be/vYggjakKYwo

[27] https://mainline.i3s.unice.fr/Wasabi-Pedalboard/

[9] F. L. Schiavoni, L. L. Gonçalves and A. L. N. Gomes, "Web Audio application development with Mosaicode," in *Proceedings of the Brazilian Symposium on Computer Music*, São Paulo, 2017.

[10] J. O. Smith III, "Audio signal processing in Faust," *online tutorial: https://ccrma. stanford. edu/jos/aspf,* 2010.

[11] Y. Orlarey, D. Fober and S. Letz, "FAUST : an Efficient Functional Approach to DSP Programming," in *New Computational Paradigms for Computer Music*, E. D. France, Ed., 2009, pp. 65-96.

[12] S. Denoux, Y. Orlarey, S. Letz and D. Fober, "Composing a Web of Audio Applications," in *Proceedings of the Web Audio Conference*, Paris, 2015.

[13] S. Letz, S. Denoux, Y. Orlarey and D. Fober, "Faust audio DSP language in the Web," in *Proceedings of the Linux Audio Conference*, Mainz, 2015.

[14] S. Ren, S. Letz, Y. Orlarey, R. Michon, D. Fober, M. Buffa, E. Ammari and J. Lebrun, "FAUST online IDE: dynamically compile and publish FAUST code as WebAudio Plugins," in *Proceedings of the Web Audio Conference*, Trondheim, 2019.

[15] R. Michon, J. Smith and Y. Orlarey, "New Signal Processing Libraries for Faust," in *Proceedings of the Linux Audio Conference*, Saint-Etienne, 2017.