Widget Modulation in FAUST

Yann Orlarey, Stéphane Letz, Romain Michon

▶ To cite this version:

Yann Orlarey, Stéphane Letz, Romain Michon. Widget Modulation in FAUST. International Faust Conference, Nov 2024, Turin, Italy. hal-04762253

HAL Id: hal-04762253
https://hal.science/hal-04762253v1

Submitted on 31 Oct 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

WIDGET MODULATION IN FAUST

Yann Orlarey
Univ Lyon, Inria, INSA Lyon, CITI, EA3720, 69621 Villeurbanne, France
yann.orlarey@inria.fr

Stéphane Letz
Univ Lyon, GRAME-CNCM, INSA Lyon, Inria, CITI, EA3720, 69621 Villeurbanne, France
letz@grame.fr

Romain Michon
Univ Lyon, Inria, INSA Lyon, CITI, EA3720, 69621 Villeurbanne, France
romain.michon@inria.fr

ABSTRACT

This article presents a novel extension to the FAUST programming language called *Widget Modulation*. Inspired by *Modular Synthesizer*, this high order operation enables developers to effortlessly implement *voltage control type* modulation to existing FAUST circuits.

Although signal modulation can easily be achieved by writing the necessary code during circuit development, *Widget Modulation* expressions enable it *a posteriori*, after the circuit has been developed and without modifying its code. This feature allows for easy *reuse and customization* without prior planning by the original circuit designer, offering a new level of expressivity and flexibility in FAUST circuit design.

1. INTRODUCTION

The development of voltage control in analog sound synthesis, beginning in the mid-20th century, marks a milestone in the history of electronic music. This technique involves using electrical voltages to modulate various synthesizer parameters, such as the pitch or amplitude of an oscillator or the cutoff frequency of a filter. This capability allows synthesis parameters to evolve, imparting timbral richness, expressiveness, and dynamism to the sounds produced.

Key figures in the development of voltage control for analog sound synthesis include Hugh Le Caine, Robert Moog, and Don Buchla. Building on Hugh Le Caine's concept of the voltage-controlled oscillator (VCO), Robert Moog established a crucial standard for modular synthesizers known as 1V/oct, where an increase of one volt corresponds to a pitch change of one octave [1].

This standard enables a form of recursivity within the synthesizer itself: the sound signals produced by one module can control the parameters of other modules, including itself. This recursive capability is a key factor in the richness and complexity of the sounds generated by modular synthesizers. Let's quote Suzanne Ciani in [2]: *"What we all love is the hands-on experience of patching and tweaking … the way it engages both our brains and our bodies, the freedom of choice it offers, the individualism, the uniqueness."*

Implementing *voltage control* principals, à la Modular Synthesizer, in FAUST [3] had always been straightforward. All we need to do is add an audio input and implement a *modulation operation* that describes how to combine this additional input signal with that of the widget we want to modulate. The modulation operation can be as simple as an addition or multiplication.

As an example, let's start with a simple oscillator: `myosc`, with a frequency control, but no modulation possibility:

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

myosc = vslider("freq[style:knob][scale:log
    ]", 440, 20, 20000,0.1)
       : os.osc;

process = myosc;
```

Let's now look at how to transform `myosc` to create a frequency modulation (FM) circuit [4]. To do this, we need to modulate the frequency of the oscillator by introducing the influence of another oscillator. Specifically, we will achieve this by adding the output signal of the second oscillator (`mymod`) to the frequency control (the "freq" widget) of the first oscillator, as in the following code:

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

myosc = +(vslider("freq[style:knob][scale:
    log]", 440, 20, 20000,0.1))
       : os.osc;

mymod = hslider("fmod[style:knob][scale:log
    ]", 110, 20, 1000,0.01)
       : os.osc * hslider("amod[style:knob
          ]", 25, 0, 1000, 0.01);

process = mymod : myosc;
```

The modification was minimal. All we had to do was add an input signal to `myosc` and sum it with the "freq" widget of the oscillator. However, we could do this because we had access to the source code of `myosc`. If `myosc` had been defined in a library, we would have had to either modify the library or duplicate the `myosc` code in our program.

As we will see, *Widget Modulation* allows us to do the same kind of transformation but without modifying the source code! It, therefore, makes FAUST's code reuse mechanisms, `library()` and `component()` even more useful.

2. EXAMPLES OF WIDGET MODULATION

Before a more formal description of *Widget Modulation*, let's consider some very simple examples using `dm.freeverb_demo` from the standard library.

Without inspecting the code, just by looking at the user interface of `dm.freeverb_demo` (figure 1) we can see the names of the various widgets that are involved and that could possibly be modulated.

Here we are interested in the `"Wet"` slider that controls the balance between the wet (reverbered) and dry (unprocessed) signals.

In order to modulate the `"Wet"` slider, we write:

Figure 1: *Freeverb user interface.*

```
["Wet" -> dm.freeverb_demo]
```

As we can see, the syntax of a *Widget Modulation* deliberately resembles that of a *lambda-abstraction*, although the semantics are quite different and the two should not be confused. Here the string `"Wet"` identifies the target of the modulation, the `vslider("Wet",...)` in the definition of `dm.freeverb_demo`. Because we didn't specify a modulation circuit, the modulation circuit is implicitly assumed to be a multiplication.

The resulting circuit has now three inputs: the new input for the modulation signal, and the original left and right inputs of the reverb. Moreover the signal delivered by the `"Wet"` slider is multiplied by the input modulation signal everywhere in the reverb circuit.

This extra input can now be connected to an oscillator to modulate the `Wet` parameter as in:

```
1+os.osc(0.1)/4,_,_: ["Wet" -> dm.
    freeverb_demo];
```

Here the modulation signal `1+os.osc(0.1)/4` is an oscillator with a frequency of 0.1 Hz, and an amplitude of 0.25. The `1+` is used to ensure that the modulation signal is between `0.75` and `1.25`.

Modulation Circuit. In the previous example, we didn't indicate a modulation circuit. To do so, we use the symbol `':'` followed by the modulation circuit. For example `"Wet":+` indicates the use of an addition as a modulation circuit. It means that our previous example is equivalent to:

```
1+os.osc(0.1)/4,_,_: ["Wet":* -> dm.
    freeverb_demo];
```

By writing `"Wet":*` we explicitly stated to use a multiplication `*` as a modulation circuit. Please note that the `:` symbol in `"Wet":*` is used to separate the widget name from the modulation circuit and should not be confused with the sequential composition operator `:`, even if it also suggests an idea of connection.

We'll come back to this later, but a modulation circuit can be of three types. A circuit with *two inputs and one output*, like `*` or `+`; a circuit with *one input and one output*, like `*(2)`; or a circuit with *no input and one output*, like `0.75`.

1. Only a modulation circuit with two inputs, like `+` or `*` creates an external modulation input. Its first input is connected to the widget, and the second one becomes the modulation input.

2. Another possibility is to describe the entire modulation circuit in a single expression, in which case there is no need for an additional input, as in the following example:

```
["Wet":*(1+os.osc(0.1)/4) -> dm.
    freeverb_demo];
```

3. Finally, we can completely replace the target widget with a modulation circuit that has no inputs, for example:

```
["Wet":0.75 -> dm.freeverb_demo];
```

Then the slider will be removed from the user interface and replaced by a constant value of 0.75, with potential speed up of the computation.

Instead of replacing a widget with a constant, we can replace it with another widget, for example to change its name, style, range, etc.:

```
["Wet":vslider("WetDry", 0.25, 0, 1,
    0.01) -> dm.freeverb_demo];
```

In this case, all occurrence of `vslider("Wet", 0.25, 0, 1, 0.01)` in `dm.freeverb_demo` is replaced by `vslider("WetDry", 0.25, 0, 1, 0.01)`.

Multiple Targets. In the previous examples, we only had one target widget. We can specify more than one by separating them with commas as in the following example:

```
["Wet", "Damp", "RoomSize" -> dm.
    freeverb_demo]
```

The resulting circuit has five inputs, three modulation inputs and two reverb inputs. The first input modulates the `"Wet"` widget, the second the `"Damp"` widget, and the third the `"RoomSize"` widget. These three inputs are followed by the two inputs of the reverb.

Please note that the above expression is equivalent to the "curryfied" version:

```
["Wet" -> ["Damp" -> ["RoomSize" -> dm.
    freeverb_demo]]]
```

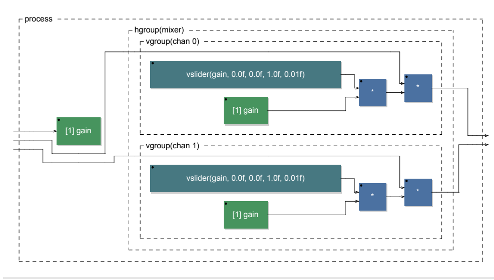

Figure 2: *All gain controls are modulated by the same input.*

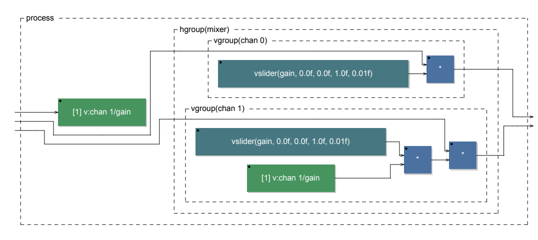

Figure 3: *Only the gain of channel 2 is modulated.*

Multiple Matches. It might happen that the same name matches multiple widgets in different groups. In this case, all the matched widgets will be modulated by the same audio input.

In the following example we have a kind of two voices mixer:

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

mixer = hgroup("mixer",
          par(i,2,
            vgroup("chan %2i",
              *(vslider("gain", 0, 0, 1,
                  0.01))
            )
          )
        );

process = ["gain" -> mixer];
```

Since in both channels we have “gain” widget, the modulation will affect both channels as we can see on the bloc-diagram figure 2.

For a more specific selection of the target widget, we can include the names of some or all of the enclosing groups of the target widget, as in `["v:chan 1/gain" -> mixer]`. Here, only the gain of channel number 1 will be modulated (see figure 3).

3. SYNTAX OF *WIDGET MODULATION*

In the preceding examples, we have provided an informal overview of *Widget Modulation*, aiming to offer a relatively intuitive understanding. Now, we will present a more formal description using syntactic rules in Backus-Naur Form (BNF), starting from the *Widget Modulation* expression itself:

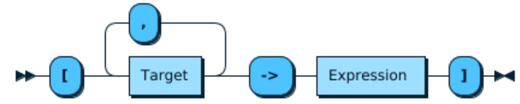

Figure 4: *WidgetModulationExpression.*

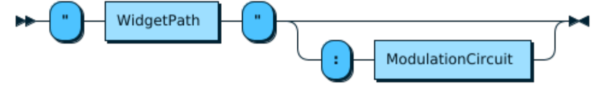

Figure 5: *Target.*

3.1. WidgetModulationExpression

```
WidgetModulationExpression
         ::= '[' Target ( ',' Target )*
             '->' Expression ']'
```

A *Widget Modulation* expression is composed of a list of target widgets and a modulated expression in which the target widgets are presumably used. The targets are separated by a comma sign (`,`). There must be at least one target. A *Widget Modulation* without targets is not allowed. The targets and the modulated expression are separated by the sign `->`, and the whole *Widget Modulation* expression is enclosed in square brackets.

3.2. Target

```
Target   ::= '"' WidgetPath '"'  ( ':'
    ModulationCircuit )?
```

A *Target* is composed of a *WidgetPath* that identifies the widget to modulate, and an optional *ModulationCircuit* that indicates how to combine the signal delivered by the widget with the modulation signal. If no ModulationCircuit is provided, the default is multiplication.

3.3. WidgetPath

```
WidgetPath
         ::= ( ( 'h:' | 'v:' | 't:' )
             GroupLabel '/' )* WidgetLabel
```

The *WidgetPath* is used to identify widgets in a modulated expression. It is a string composed of a widget label, optionally preceded by a sequence of group labels separated by slashes. The widget label is matched after removing any metadata. Group labels are used to disambiguate the widget to match, but they do not have to be consecutive.

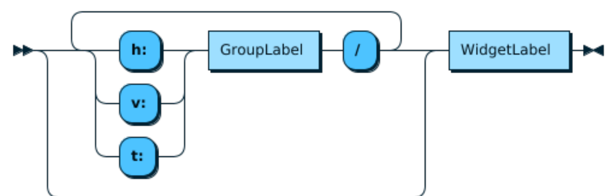

Figure 6: *WidgetPath.*

3.4. ModulationCircuit

The *ModulationCircuit* describe how to modulate the signal produced by the target widget. It can be any FAUST circuit with up to two inputs and one output. Three cases have to be considered:

- $(2 \rightarrow 1)$: A binary circuit with two inputs and one output, for example `+` as in `"Wet":+`. In this case, and only in this case, an additional input is created and the value of the widget is combined into the binary circuit before being used in the modulated expression.
- $(1 \rightarrow 1)$: A unary circuit with one input and one output, for example `*(lfo(10, 0.5))` as in `"Wet":*(lfo(10, 0.5))`. In this case, the value of the widget is routed into the unitary circuit before being used in the modulated expression, and no additional input is created.
- $(0 \rightarrow 1)$: A constant circuit with no input and one output, for example `0.75` as in `"Wet":0.75`. In this case all the occurrences of widget are simply removed and replaced by the constant circuit in the modulated expression. This is convenient to simplify a rich interface when some widgets that are not needed. In this case, no additional input is created.

3.5. Compilation

Widget Modulation is a circuit transformation primitive that is handled in the first phase of the compilation process. During this phase the FAUST program is evaluated to produce a circuit in *normal form* (a *flat* circuit composed solely of interconnected primitives).

Let's illustrate this process on our previous example:

```
["Wet":*(1+os.osc(0.1)/4) -> dm.
    freeverb_demo]
```

The compiler first evaluates `*(1+os.osc(0.1)/4)` and `dm.freeverb_demo` into their respective normal forms `c1` and `c2`. It then computes the normal form of `["Wet":*(1+os.osc(0.1)/4)-> dm.freeverb_demo]` by replacing every widget `w` labeled `"Wet"` occurring in `c2` with `w:c1`.

As a circuit transformation, *Widget Modulation* represents a new type of operation for FAUST, distinct from circuit composition operations (`:`, `,`, `\~`, `<:`, `:>`), which assemble existing circuits without transforming them. Despite this distinction, *Widget Modulation* fully aligns with the philosophy of a programming language dedicated to the description and implementation of audio circuits.

4. EXAMPLES OF MODULATION CIRCUITS

The ability to specify our own modulation circuits provides a lot of flexibility and expressiveness to *Widget Modulation*. Here, we give some examples of modulation circuits, some of which exploit the fact that it is possible to know the minimum and maximum values of a signal using the primitives `lowest` and `highest`, thereby ensuring that the signal after modulation remains within the widget's limits.

4.1. Frequency Modulation

Let's start with a simple example of frequency modulation showing the usage of simple additive and multiplicative modulations.

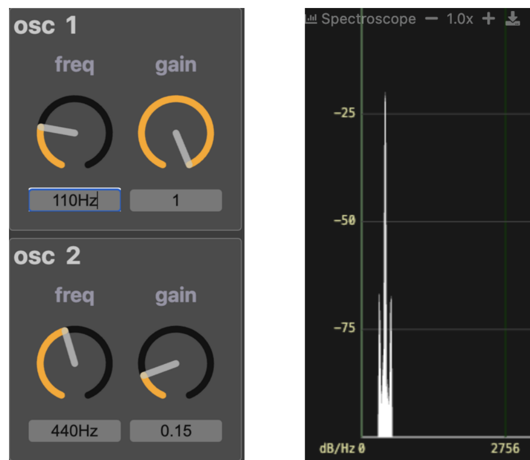

Figure 7: *Simple frequency modulation.*

We first define an oscillator with its own user interface consisting of two widgets, one controlling its frequency and the other its amplitude.

```
osc(n) = hgroup("osc %2n", os.osc(f) * g
    with {
     f = vslider("freq[scale:log][style:knob
         ][unit:Hz]",440,0.25,20000,1);
     g = vslider("gain[style:knob]", 0, 0,
         1, 0.01);
});
```

The `n` parameter is used in the group label to distinguish oscillators, so that we can use more than one. The minimal value for the frequency, 0.25 Hz, is deliberately outside the audible range in order to use the oscillator also as a LFO.

Let's look at a first example of frequency modulation using an addition as the modulation circuit:

```
process = osc(1) : ["freq":+ -> osc(2)];
```

The user interface and resulting spectrum are shown in figure 7. We recognize a FM spectrum, but the amplitude of the modulation oscillator is not high enough to obtain a rich spectrum. We can fix the problem by amplifying the modulation signal:

```
process = osc(1)*500 : ["freq":+ -> osc(2)
    ];
```

This gives us the spectrum figure 8.

Now let's add a third modulation stage, to modulate the "gain" of oscillator 1 and obtain a periodic variation in the spectrum:

```
process = osc(0)+1 : ["gain" -> osc(1)*500]
     : ["freq":+ -> osc(2)];
```

The resulting program can be try on line here

4.2. Advanced Modulation Circuits

In the previous examples, we did not account for the possibility of the modulated widget signal exceeding the limits of the initial widget. However, there are scenarios where this is important. Therefore, we now present several more advanced modulation circuits that ensure that the output value respects the range of values of

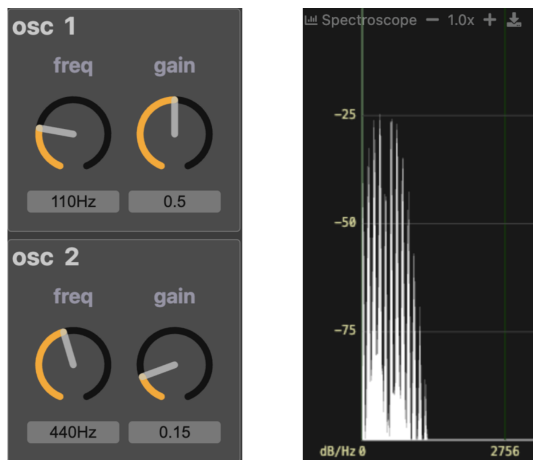

Figure 8: *Improved FM circuit.*

the widget w. As a reminder, we can determine the minimum and maximum values of a signal using the lowest and highest primitives.

addp and add modulations. The principle here is to add a value taken between two limits to the widget. A distinction is made between two cases, depending on the nature of the modulation signal. It could be a positive modulation signal, between 0 and +1, such as the signal from an envelope follower. Alternatively, it could be an audio signal, between -1 and +1, such as the signal from an oscillator. The obtained value is then clipped to the limits of the widget.

The first function, addp, adds to the widget w a value between v1 and v2 based on a positive modulation signal m ranging from 0 to +1:

```
addp(v1,v2,w,m) = max(lo, min(hi, w + v))
  with {
    lo = lowest(w);
    hi = highest(w);
    v = v1+m*(v2-v1);
  };
```

The second function, add, adds to the widget w a value between v1 and v2 based on an audio modulation signal m ranging from -1 to +1.

```
add(v1,v2,w,m) = addp(v1,v2,w,(m+1)/2);
```

mulp and mul modulations. The first function mulp multiply the widget value w by a factor between f1 and f2 based on a positive modulation signal m ranging from 0 to +1.

```
mulp(f1,f2,w,m) = max(lo, min(hi, w * f))
  with {
    lo = lowest(w);
    hi = highest(w);
    f = f1+m*(f2-f1);
  };
```

The second one mulp multiply the widget value w by a factor between f1 and f2 according to an audio modulation signal m ranging from -1 to +1.

```
mul(f1,f2,w,m) = mulp(f1,f2,w,(m+1)/2);
```

mapp and map modulations. These last two functions allow you to completely replace a widget (causing it to disappear from the user interface) with a value that varies between two bounds controlled by a modulation signal. The first function mapp replaces the widget by a value between v1 and v2 based on a positive modulation signal p ranging from 0 to +1.

```
mapp(v1,v2,w,p) = v1 + p*(v2-v1);
```

4.3. Revisiting the Frequency Modulation Example

We can apply these new functions to revisit our frequency modulation example. Let's start by defining an md environment containing all our modulation functions:

```
md = environment {
    addp(v1,v2,w,m) = max(lowest(w), min(
       highest(w), w + v))
    with {
        v = v1+m*(v2-v1);
    };

    mulp(f1,f2,w,m) = max(lowest(w), min(
       highest(w), w * f))
    with {
        f = f1+m*(f2-f1);
    };

    mapp(v1,v2,w,p) = v1 + p*(v2-v1);

    add(v1,v2,w,m) = addp(v1,v2,w,(m+1)/2);
    mul(f1,f2,w,m) = mulp(f1,f2,w,(m+1)/2);
    map(v1,v2,w,m) = mapp(v1,v2,w,(m+1)/2);

};
```

The revised frequency modulation example is as follows:

```
process = osc(0)
        : ["freq":md.add(-500,500) -> osc
            (1)];
```

It is interesting to note that the same target widget can be modulated several times by different modulation circuits. In the following *Widget Modulation* expression, two modulation circuits are applied to the same "freq" widget. It is first modulated by an add(-600,600), and the result by a mul(0.1,10):

```
["freq":md.add(-600,600), "freq":md.mul
    (0.1,10) -> osc(1)]
```

The circuit figure 9, shows how this double modulation is implemented.

To complete this section, here is a more elaborate example. It combines double frequency modulation—by an oscillator and by its own output signal via feedback—with the modulation of the "gain" widgets of these two oscillators by a third oscillator.

```
process = osc(0) : ["gain":md.add(0,0.5) ->
     (_ ,osc(1): ["freq":md.add(-600,600), "
    freq":md.mul(0.1,10) -> osc(2)])~@(200)
    ]<: _,@(5000);
```

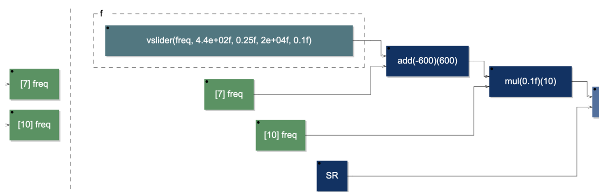

Figure 9: *Double modulation of the* `"freq"` *widget.*

5. CONCLUSION

This article introduces *Widget Modulation*, a novel extension to the FAUST programming language. This high-order primitive directly manipulates audio circuits, marking the first instance of such functionality within FAUST.

Inspired by the principles of modular synthesizers, *Widget Modulation* enables developers to seamlessly implement voltage control-type modulation into existing FAUST circuits. This allows for the redesign of user interfaces without necessitating direct access to the underlying source code.

While mastering the use of *Widget Modulation* may require time, its potential to significantly influence the development of FAUST libraries is substantial [5]. Users will be empowered to create libraries of modules, akin to those found in modular synthesizers, featuring rich and detailed user interfaces, with the assurance that a posteriori customization remains feasible. Furthermore, the extension opens avenues for the development of new libraries dedicated to modulation circuits.

6. REFERENCES

[1] Laurent de Wilde, *Les fous du son, d'Edison à nos jours*, Grasset, 2016.

[2] Suzanne Ciani, *Foreword*, Bjooks, May 2018.

[3] Yann Orlarey, Stéphane Letz, and Dominique Fober, *New Computational Paradigms for Computer Music*, chapter "Faust: an Efficient Functional Approach to DSP Programming", Delatour, Paris, France, 2009.

[4] John M. Chowning, "The synthesis of complex audio spectra by means of frequency modulation," *Journal of the audio engineering society*, vol. 21, no. 7, pp. 526–534, 1973.

[5] Julius O. Smith, "Signal processing libraries for Faust," in *Proceedings of Linux Audio Conference (LAC-12)*, Stanford, USA, May 2012.