Kabelsalat: Live Coding Audiovisual Graphs on the Web and Beyond

Felix Roos
Unaffiliated
flix91@gmail.com

ABSTRACT

This paper introduces Kabelsalat, a graph-based live coding environment that targets multiple platforms and languages. It works by translating a Domain Specific Language (DSL) into a signal flow graph, to finally be compiled into a sequence of instructions optimized for real time signal processing. The DSL has been implemented both in JavaScript and Lua. Currently, the compiler output can either be optimized JavaScript to run in the browser, or optimized C code to run natively. Being able to add other output languages in the future is part of the design. The Browser version includes a REPL that implements a range of audio DSP nodes that are reminiscent of a modular synthesizer. Notable features include single sample feedback and multi-channel expansion, inspired by SuperCollider. The core module of kabelsalat has also been used to implement a stripped down version of the Hydra video synthesizer, applying the same principles to generate GLSL code. In the future, kabelsalat might become an alternative audio engine of strudel, offering more sound design capabilities, compared to the current superdough engine, which uses the browser's built-in Web Audio Nodes.

1 Introduction

Graphs are predominantly used to represent the signal flow of live coding systems, such as SuperCollider (McCartney 2002), Glicol (Lan and Jensenius 2021), Genish.js (Roberts 2017), Hydra (Jack 2018), and Punctual (Ogborn 2018). They are also ideal for patcher interfaces, such as Pure Data (Puckette 1996), NoiseCraft (Chevalier-Boisvert 2021) and cables.gl (Kombuechen 2020). In Web Audio specifically, there has been a shift from using the built-in Web Audio nodes to the more recently introduced AudioWorklets (Choi 2018; Roberts 2018). AudioWorklets allow writing signal processors that are self-contained, without being too intertwined with web platform specifics. Furthermore, they can be used to implement DSP algorithms that rely on single-sample processing, which is not possible in the block based processing model of the Web Audio API graph (Roberts 2017). These properties were the initial motivation to build an audio engine that could eventually be used in Strudel (*Strudel: Live Coding Patterns on the Web* 2023). For the time being, Strudel uses the more limited built-in Web Audio graph for its audio engine called *superdough*.

Kabelsalat implements a Domain Specific Language (DSL) to represent and compile graphs suitable for single-sample processing. The compilation strategy, as well as many audio nodes of Kabelsalat are based on the browser-based NoiseCraft (Chevalier-Boisvert 2021) synthesizer. The NoiseCraft compiler was rewritten to a version that encapsulates its core logic from the output language, allowing each node to control its own code generation. With this addition, the core language is not specifically tied to audio graphs or a single output language. To test the viability of compiling to another language, Kabelsalat can also compile a graph to C code, to be played as a standalone binary. As another proof-of-concept, Kabelsalat was used to compile Hydra (Jack 2018) patches to GLSL code, showing how the same concepts apply in the visual domain.

2 Introducing the Kabelsalat DSL

The following example shows how a simple subtractive synthesis patch can be written in Kabelsalat:

```
// sawtooth wave at 55Hz:
saw(55)
  // modulated low-pass-filter
  .lpf(sine(1).range(0.4, 0.8))
  // modulated amplitude:
  .mul(sine(4).range(0.25, 1))
  // send to output:
  .out();
```

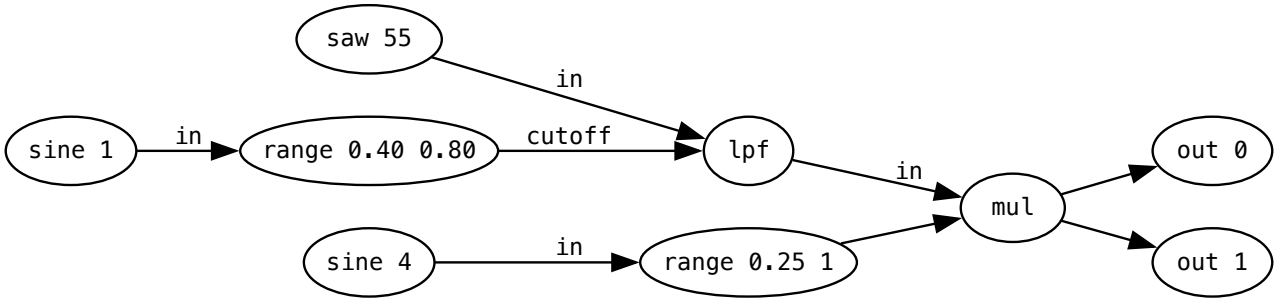

Figure 1: Graph visualization of a sawtooth wave with filter and amplitude modulation

This type of syntax corresponds metaphorically to the operation of a modular synthesizer, whose signal generation and processing is organized into several modules, represented by function calls. Each function call creates a node, given its inputs as arguments. Each argument can be either a constant value or another node, which allows creating patches of arbitrary depth. Figure 1 shows the example's graphical representation, which was created by Kabelsalat's visualizer.

2.1 Method Chaining

As demonstrated in the previous example, Kabelsalat employs method chaining to "flatten" the syntax. The same example can be expressed without method chaining as well:

```
// send to output:
out(
  // modulate amplitude
  mul(
    // modulated low-pass-filter
    lpf(
      // sawtooth wave at 55Hz:
      saw(55),
      range(sine(1), 0.4, 0.8)
    ),
    range(sine(2), 0.25, 1)
  )
);
```

This notation only consists of function calls, which is syntactically simpler, but arguably more difficult to parse as a human. Compared to the method chaining example, the expression is deeply nested, with a wider distance between logically grouped tokens. Additionally, editing the expression involves a lot of extra indenting and cursor movement. To avoid these difficulties, Kabelsalat includes method chaining, similar to Hydra (Jack 2018) and Strudel (*Strudel: Live Coding Patterns on the Web* 2023). Method chaining can be seen as a way to write expressions in the same order as infix notation, without the need to overload operators. When a method is called on a node, that node is used as the first input of the method.

2.2 Multichannel Expansion

Inspired by SuperCollider, Kabelsalat allows expanding the channels of a graph by passing an Array to a node:

```
// create two channels of filtered sawtooth waves
saw([200, 300]).lpf(0.5).out([0, 1]);
```

Multichannel expansion (Liljedahl 2014) simplifies writing and editing graphs that represent multiple signal channels. It allows generating large graphs with relatively few characters. In Kabelsalat, it works as follows: The Array is interpreted as a special `poly` node, where each element of the Array is an input. When a node receives a `poly` node with `n` inputs, `n` copies of the node are created. Each copy receives one of the values in the Array. The copied nodes are fed into a new `poly` node, which is propagated down the graph. The `poly` node will eventually end up at the bottom of the graph, where

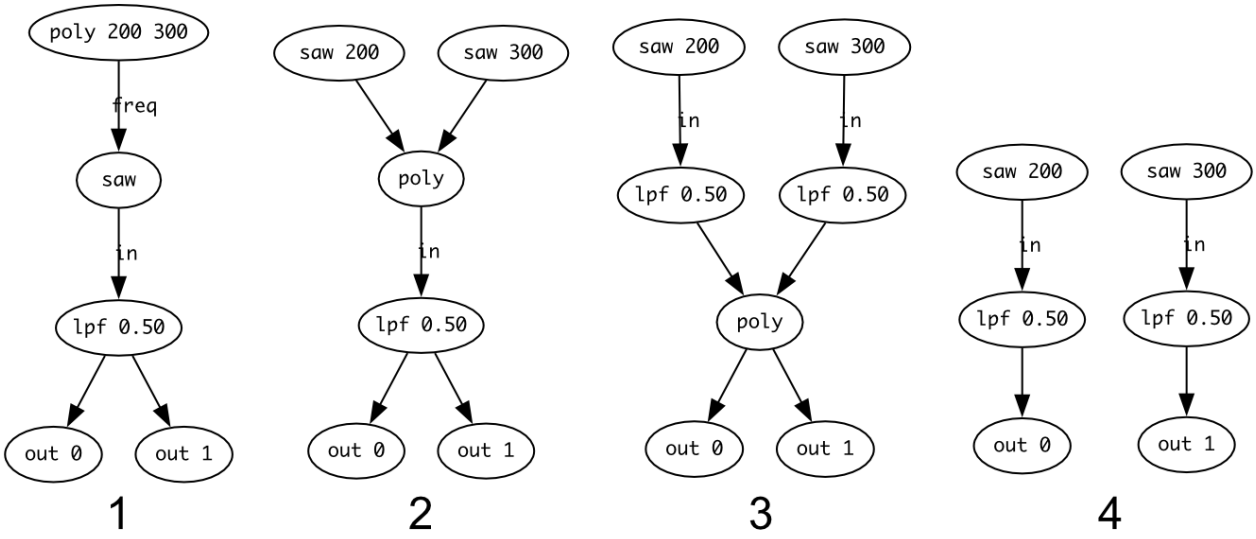

Figure 2: Multichannel Expansion Example

each channel is assigned to one `out` node. In cases where a node receives multiple `poly` nodes, the `poly` node with the most inputs determines the number of copies. The inputs of the other `poly` nodes wrap around.

Figure 2 shows a graphical version of how the `poly` node is propagated in the above example.

The end result of the expansion is equivalent to:

```
// the above is equivalent to:
saw(200).lpf(0.5).out(0);
saw(300).lpf(0.5).out(1);
```

2.3 Feedback

A feedback loop is created when a node uses its own output as an input, creating a cycle in the graph. For both audio and video synthesis, feedback plays an important role for a variety of techniques (Roberts 2017). Kabelsalat supports single sample feedback, which can be notated in 2 ways:

1. Passing an anonymous function as a function argument
2. Using the `src` node to read from an output

2.3.1 Feedback with Anynomous Functions

An anonymous function can be passed to any node to create a cycle:

```
impulse(1)
  .add((x) => x.delay(0.2).mul(0.8))
  .out();
```

In this example, the `add` node receives an anonymous function as its input, which receives its own output as an argument. This allows notating the transformsations of the feedback loop within that function. Figure 3 shows a graph visualization of this example.

2.3.2 Feedback with `src` Node

Instead of using an anonymous function, feedback can also be created with the dedicated `src` node:

```
impulse(1).add(src(0).delay(0.1).mul(0.8)).out();
```

This syntax is inspired by Hydra (Jack 2018), where feedback is also created using `src` and `out` nodes.

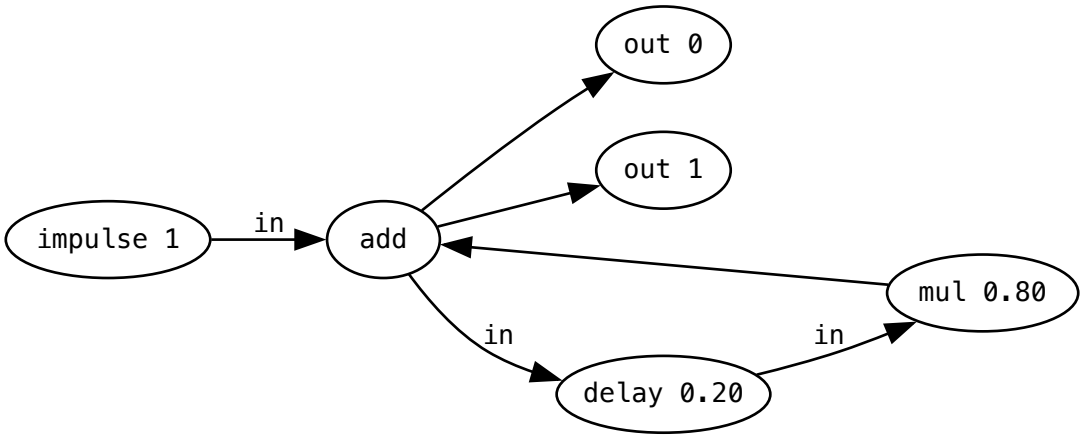

Figure 3: Graph visualization of a simple feedback delay

3 Graph Compilation

After a brief introduction of the language, this chapter explains how it is compiled into a representation that is optimized to run within the constraints of a real-time system. Note that the following examples are specifically targetting audio synthesis in JavaScript. Compilation to C or GLSL follows similar principles.

3.1 From DSL to Graph

When the DSL is evaluated, a directed graph is created. As a TypeScript interface, the structure of a `Node` can be described as:

```
interface Node {
  type: string;
  ins: Array<Node | number>;
}
```

Each `Node` has a type and an Array of inputs called `ins`. Elements inside `ins` are either other instances of `Node` or constant numeric values. Here is an example of a `Node` instance representing a filtered sawtooth wave:

```
{
  "type": "out",
  "ins": [{ "type": "lpf", "ins": [{ "type": "saw", "ins": [200] }, 0.5] }, 0]
}
```

Note that the above data is represented as json only for the purpose of readability. The actual implementation uses JavaScript Objects, where each `Node` is only referenced, meaning reused `Node` instances will not be copied. For cyclical graphs, a JSON representation does not exist, because it would create an infinite loop.

3.2 From Graph to Output Language

To generate efficient runtime code, the graph is converted into a sequence of steps. Before compilation, `Nodes` are sorted topologically, making sure each `Node`'s inputs are computed first. The compiler output for the graph of the last example is as follows:

```
r[1] = nodes[0].update(200); /* saw */
r[3] = nodes[1].update(r[1], 0.5, 0); /* lpf */
o[0] = r[3]; /* out 0 */
```

In an AudioWorklet (Choi 2018), this code will run once for each sample, typically at 44,1kHz or 48kHz. Each `Node` corresponds to one line in the generated code. It expects the following variables to be defined:

- `nodes`: instances of stateful nodes
- `r`: node value registers
- `o`: output channel registers

3.2.1 Stateful Nodes

The `nodes` Array contains instances of stateful signal processors, which are expected to be provided to the compiled function. Stateful nodes are essential for many audio DSP techniques, for example to keep track of the phase of an oscillator while its frequency is being modulated. Each audio processor needs to implement an `update` method to compute the next sample based on its input arguments. A simple sawtooth wave can be implemented in JavaScript as follows:

```
class SawOsc {
  constructor() {
    this.phase = 0;
  }
  update(freq) {
    this.phase += SAMPLE_TIME * freq;
    return (this.phase % 1) * 2 - 1;
  }
}
```

In the C language, a similar pattern can be implemented with an update function operating on a struct. In GLSL, nodes are stateless due to the parallel nature of graphics rendering.

3.2.2 Value Registers

The `r` Array contains the latest output values of each `Node`. When a graph contains cycles, the node that receives the feedback depends on a node that has not been calculated yet. By saving each `Node`'s result into the `r` Array, those nodes will automatically receive the value from the previous iteration. To illustrate this point, here is the compiled output of Figure 3:

```
r[1] = nodes[0].update(1); /* impulse */
r[3] = nodes[1].update(r[6], 0.2); /* delay */
r[5] = r[3] * 0.8; /* mul */
r[6] = r[1] + r[5]; /* add */
o[1] = r[6]; /* out 1 */
o[0] = r[6]; /* out 0 */
```

In Line 2, `r[6]` references the value of the previous iteration, making feedback possible.

3.2.3 Output Registers

The `o` Array keeps track of each output channel. After each iteration of the compiled sequence, `o[0]` and `o[1]` can be passed to the sound card for playback. The `out` function of the DSL takes a channel as its only argument, which falls back to `[0,1]`. This ensures both stereo channels receive a value by default.

3.3 Node Compilation

To encapsulate the compiler logic from the output language, each node definition contains a `compile` function that is expected to output its target language. The compiler's sole responsibility is to pass the correct register names and constant values to the compile function. An `impulse` node could be defined to output C code as:

```
let saw = registerNode("impulse", {
  ugen: "ImpulseOsc",
  compile: ({ vars: [freq = 0], name, node, ugen }) =>
    `${name} = ${ugen}_update(${node},${freq}); /* ${ugen} */`,
});
```

In comparison to the JavaScript version, the C version of Figure 3 is:

```
r[1] = ImpulseOsc_update(nodes[0],1); /* ImpulseOsc */
r[3] = Delay_update(nodes[1],r[6],0.2); /* Delay */
r[5] = r[3] * 0.8;
r[6] = r[1] + r[5];
o[0] = r[6]; /* out 0 */
o[1] = r[6]; /* out 1 */
```

References

Chevalier-Boisvert, Maxime. 2021. "NoiseCraft." https://github.com/maximecb/noisecraft.

Choi, Hongchan. 2018. "Audioworklet: The Future of Web Audio." In *International Conference on Mathematics and Computing*. https://api.semanticscholar.org/CorpusID:69755393.

Jack, Olivia. 2018. "Hydra." https://github.com/ojack/hydra.

Kombuechen, Thomas. 2020. "Cables.gl." https://github.com/maximecb/noisecraft.

Lan, Qichao, and Alexander Refsum Jensenius. 2021. "Glicol: A Graph-Oriented Live Coding Language Developed with Rust, WebAssembly and AudioWorklet." In *Proceedings of the International Web Audio Conference*, edited by Luis Joglar-Ongay, Xavier Serra, Frederic Font, Philip Tovstogan, Ariane Stolfi, Albin A. Correya, Antonio Ramires, Dmitry Bogdanov, Angel Faraldo, and Xavier Favory. WAC '21. Barcelona, Spain: UPF.

Liljedahl, Jonatan. 2014. "Multichannel Expansion Documentation." https://doc.sccode.org/Guides/Multichannel-Expansion.html.

McCartney, James. 2002. "Rethinking the Computer Music Language: SuperCollider." *Computer Music Journal* 26 (4): 61–68. https://doi.org/10.1162/014892602320991383.

Ogborn, David. 2018. "Hydra." https://github.com/dktr0/Punctual.

Puckette, Miller. 1996. "Pure Data: Another Integrated Computer Music Environment."

Roberts, Charles. 2017. *Strategies for Per-Sample Processing of Audio Graphs in the Browser*. Web Audio Conference.

———. 2018. *Metaprogramming Strategies for AudioWorklets*. Web Audio Conference.

Strudel: Live Coding Patterns on the Web. 2023. Zenodo. https://doi.org/10.5281/zenodo.7842142.