Kabelsalat: Live Coding Audiovisual Graphs on the Web and Beyond

Felix Roos
Unaffiliated
flix91@gmail.com

Raphaël Maurice Forment
Université Jean Monnet
raphael.forment@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. This graph can be compiled into a sequence of instructions optimized for real time signal processing. The DSL has been implemented both in JavaScript and Lua. The compiler can either output JavaScript code to run in the browser or optimized C code to run natively. The possibility of adding other target languages is an integral part of KabelSalat's design. The browser version includes a REPL and features a range of audio DSP nodes reminiscent of modular synthesizers. Notable features include single sample feedback and multi-channel expansion inspired by the SuperCollider audio engine. The core module of Kabelsalat has also been used to implement a stripped down version of the Hydra video synthesizer, thus demonstrating that the same underlying principles can be adapted both for audio and video generation. In the future, KabelSalat might become an alternative audio engine for 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 often used to represent the signal flow of live coding systems, as demonstrated by Glicol (Lan and Jensenius 2021), Genish.js (Roberts 2017), Hydra (Jack 2018) or Punctual (Ogborn 2018). This type of representation is also common for live patching environments such as NoiseCraft (Chevalier-Boisvert 2021), cables.gl (**cables.gl?**) or VCVRack. In this context, graphs often feel more natural, as they allow for a more direct and graphical representation of the signal flow. Many important audio programming languages from the past decades, such as Pure Data (Puckette 1996) and SuperCollider (McCartney 2002), are also computing audio based on the concept of signal flow graphs. Graphs often allow for an optimized execution of the signal processing chain, as they can be analyzed and optimized before execution.

Nowadays, this dataflow paradigm is getting increasingly more common in the context of web audio. One can notice a gradual shift from the usage of built-in Web Audio nodes[1] to the more recently introduced AudioWorklets (Choi 2018; Roberts 2018). AudioWorklets allow the creation of self-contained signal processors, less dependant on web platform specifics. These can be developed in JavaScript and/or compiled to WebAssembly from any other language supporting this compilation target. Furthermore, they introduce a significant advantage over built-in web audio nodes. Audioworklets can be used to implement DSP algorithms that rely on single-sample processing. This property offers many advantages over block-based processing such as the possibility of writing audio feedback loops or granular synthesis algorithms (Roberts 2017). Single-sample processing also removes the block size constraint of web audio nodes, which can be a constraint for some algorithms (EXAMPLES).

The creative exploration of the technical possibilities offered by AudioWorklets was 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*[2].

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 an application of the same concepts in another domain.

1

2

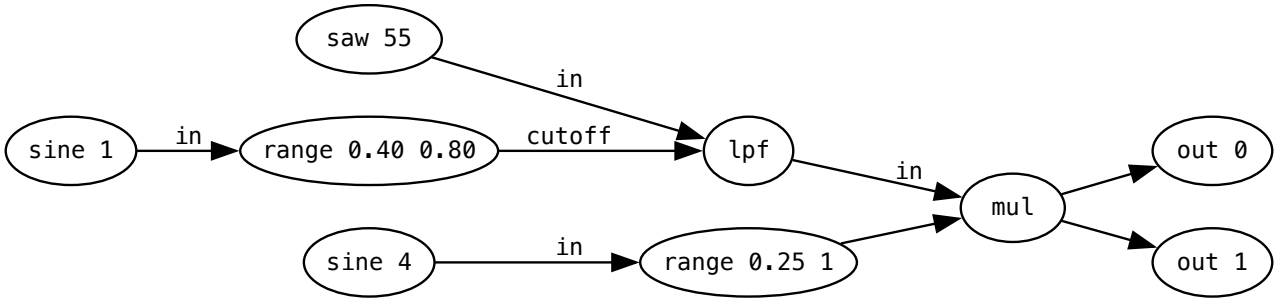

Figure 1: Graph visualization of a low-pass filtered sawtooth oscillator with amplitude modulation. KabelSalat is able to generate a graphical representation of the audio graph in real time using GraphViz[4].

2 Introducing the Kabelsalat DSL

KabelSalat provides a terse and practical syntax for writing audio graphs on-the-fly. Moreover, many syntactic shortcuts have been developed to make the code more readable while also being easier to write. The following example, as a first introduction, shows how a simple subtractive synthesis patch can be written:

```
// 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();
```

The syntax relies heavily on method chaining, as a way to emulate the operation of a modular synthesizer. Each function or method call represents a signal generator or processor, which can be connected to other modules through chaining or reference. Each function or method creates a node, whose arguments can be either constant values or other nodes. This allows creating patches of arbitrary depth and complexity.

2.1 Method Chaining

The previous example implicitely demonstrates how KabelSalat "flattens" the syntax in order to make the code more readable. The method chaining pattern is used to connect nodes in a linear fashion, which is more intuitive than nested function calls. A similar strategy was used for the composition of functional musical patterns in Strudel (*Strudel: Live Coding Patterns on the Web* 2023).

The following example shows how the same patch could have been written without method chaining:

```
// 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)
  )
);
```

[4]

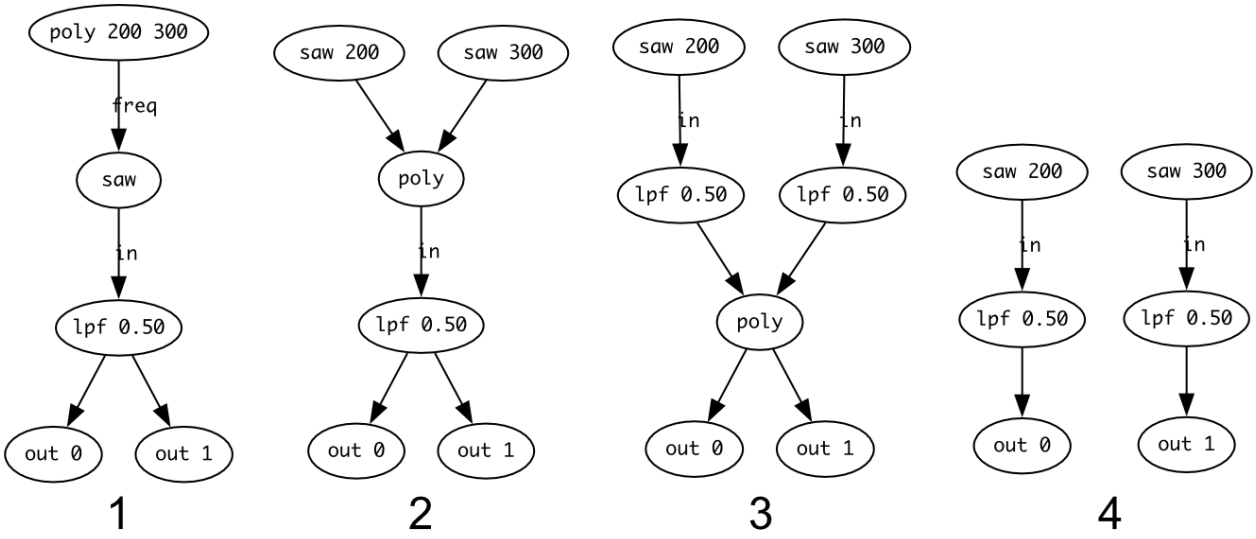

Figure 2: Multichannel Expansion Example

This notation only consists of function calls, which is syntactically simpler, but arguably more difficult to parse for the musician. 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

KabelSalat borrows the concept – and the syntax – of multichannel expansion from SuperCollider[5] (Liljedahl 2014). This feature allows the duplication of a node or a chain of nodes to multiple channels. Large audio graphs can thus be generated with relatively few characters. A node can receive an Array as input:

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

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);
```

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 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.

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:

[5]Link to documentation page.

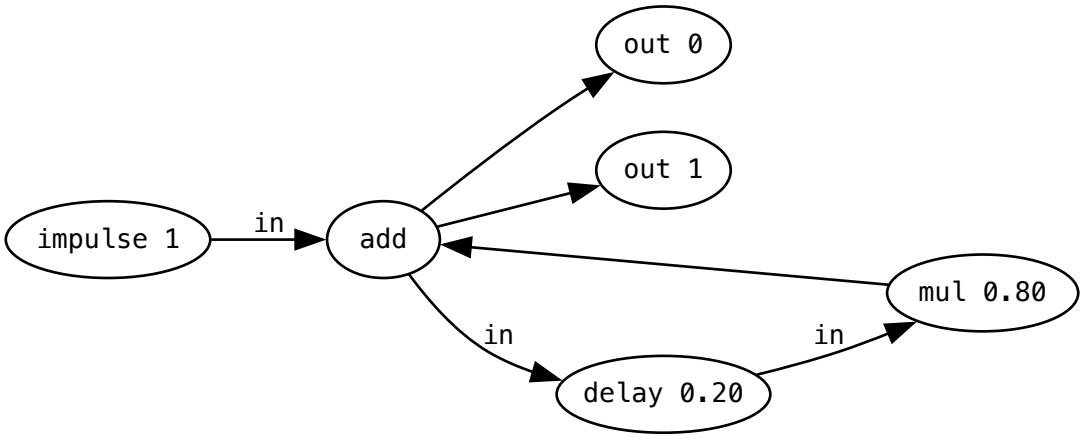

Figure 3: Graph visualization of a simple feedback delay

1. by passing an anonymous function as a function argument
2. by sing the special `src` node to read from an output.

2.3.1 Feedback loop using 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 loop using the source (`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.

3 Audio Graph Compilation

KabelSalat audio graphs are compiled into a representation optimized to run with good performances in the constraints of a real-time system. For the sake of demonstration, we are going to focus on the JavaScript output, but the same principles apply to the C or the GLSL targets.

3.1 First step: from DSL to Graph

When a KabelSalat script is evaluated, the DSL is parsed into a directed 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 Second step: 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 implement 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 */
```

Musical examples

4 Acknowledgements

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.

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.