A signal engine for a live coding language ecosystem

Article (Accepted Version)

Bernardo, Francisco, Kiefer, Chris and Magnusson, Thor (2020) A signal engine for a live coding language ecosystem. *Journal of the Audio Engineering Society*, 68 (10). pp. 756-766. ISSN 0004-7554

This version is available from Sussex Research Online: <http://sro.sussex.ac.uk/id/eprint/94382/>

This document is made available in accordance with publisher policies and may differ from the published version or from the version of record. If you wish to cite this item you are advised to consult the publisher's version. Please see the URL above for details on accessing the published version.

Copyright and reuse:

Sussex Research Online is a digital repository of the research output of the University.

Copyright and all moral rights to the version of the paper presented here belong to the individual author(s) and/or other copyright owners. To the extent reasonable and practicable, the material made available in SRO has been checked for eligibility before being made available.

Copies of full text items generally can be reproduced, displayed or performed and given to third parties in any format or medium for personal research or study, educational, or not-for-profit purposes without prior permission or charge, provided that the authors, title and full bibliographic details are credited, a hyperlink and/or URL is given for the original metadata page and the content is not changed in any way.

A Signal Engine for a Live Coding Language Ecosystem*

FRANCISCO BERNARDO, CHRIS KIEFER AND THOR MAGNUSSEN,

(F.Bernardo@sussex.ac.uk) (C.Kiefer@sussex.ac.uk) (T.Magnussen@sussex.ac.uk)

Experimental Music Technologies Lab / Department of Music, University of Sussex, Brighton, UK

This paper reports on early advances in the design of a browser-based ecosystem for creating new live coding languages, optimal for audio synthesis, machine learning and machine listening. We present the rationale and challenges when applying the Web Audio API to the design of a high-performance signal synthesis engine, using an *AudioWorklet*-based solution and refactoring our digital signal processing library *Maximilian.js*. Furthermore, we contribute with the latest advances in *Sema*, a new user-friendly playground which integrates the signal engine to empower the live coding community to design their own idiosyncratic languages and interfaces. The evaluation shows that the system runs with high reliability, efficiency and low latency.

0 INTRODUCTION

Infrastructural software or middleware (e.g. APIs, software libraries and frameworks, online services, toolkits) has become fundamental in the development and operation of other software, including end-user applications and other infrastructural software [1]. It empowers software engineering practices with modularity, reusability and abstraction principles [2]. It can also help to scale and accelerate innovation in specific application domains. Of particular interest, are opportunities that leverage on the emergence of infrastructural software for the Web, at the intersection of real-time interactive audio, digital signal processing (DSP) and machine learning [3].

Since its introduction in 2011, the Web Audio API (WAAP) [4] has powered a substantial number of libraries and applications for interactive sound and music. WAAP-powered libraries have implemented various approaches and features: from high-level control and abstractions over WAAP graphs and native nodes (e.g. *Tone.js* [5]), to abstractions with low-level digital signal processing (DSP) using optimised JS (e.g. *Genish.js* and *Gibberish.js* [6]), *asm.js*¹ (e.g. *Maximilian* [7]) and *WebAssembly* (Wasm) (e.g. *Faust* [8], *Csound* [9]). Some of these libraries which initially implemented custom nodes using *ScriptProcessorNode* (SPN), with its inherent limitations (e.g. latency, main thread interrupts), have

been shifting towards utilising *AudioWorklet* [10] (e.g. *Faust*, *Csound*, *Genish.js*). Such improvements have been used previously in advancing the design of live coding environments for the Web that enable custom low-level DSP and high-level control of musical processes [6], and that support end-user live coding language design [11].

Client-side Web-based machine learning middleware have shown recent and meaningful advances (e.g. *Tensorflow.js* [12], *Magenta.js* [13], *ml5.js* [14], *RapidLib.js* [15]). These libraries enable end-to-end client-side machine learning workflows with many benefits, including simplified deployment with “zero install,” instant access across platforms, data privacy and, in the cases of *Tensorflow.js*-based libraries, hardware acceleration. Common to these libraries has been a strong consideration for developer-friendly experience design, particularly in relation to API abstraction level and learning resources, documentation and examples. There are also significant challenges involved, concerning varying client-side hardware and the progress of the web browser ecosystem.

We believe that the recent improvements in Web technologies can afford the evolution of an ecosystem of user-defined, live coding languages that combine real-time machine learning, machine listening and audio threads. Previous attempts to integrate real-time interactive audio and machine learning in a scalable and accessible environment such as the Web, have been fraught with technical limitations of the single-threaded JavaScript (JS) environment and the Web Audio API *ScriptProcessorNode* (SPN) [15]. We contribute a novel system design for live

*To whom correspondence should be addressed Tel: +4401273 873481; e-mail: F.Bernardo@sussex.ac.uk

¹<http://asmjs.org/>

coding with a high-performance signal engine leveraging the *Maximilian* DSP library transpiled for *AudioWorklet*².

There are many compelling opportunities for bringing these technologies together through design exploration and application in computational creativity and arts. We are particularly interested in live coding approaches to music [16] and its symbiosis with creative machine learning. Live coding practitioners typically engage simultaneously in programming with a domain specific language (DSL) and other modalities, including audio and visual synthesis, instrument design, algorithmic creation, composition and performance [17]. Such approaches can provide a rich use case and substrate for the exploration of new web-based computational tools for empowering the live coding and creative technology communities. They can also facilitate pedagogical approaches to computational thinking in the context of creative and artistic practices (live coding [18], STEAM [19]), or new artistic processes and outcomes that may arise from the integration of real-time interactive DSP and ML technologies, in a scalable and accessible environment such as the web browser.

This article has the following structure: after introducing the artistic practice of live coding, we present the requirements and a proposal for an architecture that integrates signal synthesis and processing, machine learning and live coding language design for client-side Web applications. Section 3 presents the elements of our design strategy and rationale for refactoring Maximilian [7] for an *AudioWorklet*-based signal engine. Section 4 describes the implementation and evaluation of our signal engine. Section 5 discusses some of the main limitations for the *AudioWorklet* implementation and Section 6 concludes with the main takeaways and future work.

1 LIVE CODING IN THE ARTS

Live coding emerged as a musical performance practice in the early 2000s [20]. It has been characterised as involving a plethora of software tools, performance styles and modalities, including audio and visual synthesis, instrument design, algorithmic creation, composition and performance [17]. Common to all of these modalities is the activity of real-time programming with a domain-specific language (DSL), which becomes an instrument of sort and gives the performance a unique style. Such performances are often inspirational and humorous, given the physical and cognitive demands of coding in real-time to a live audience in a public setting.

Early practitioners invented their own live coding systems for musical and other types of artistic performance (e.g. [21]). In the past years, the live coding practice, which has been growing in popularity partly due to the cultural phenomenon of algoraves [22], has been also consolidating in established musical forms, practices and languages e.g. SuperCollider [23], ixi lang [24], Gibber [25], Sonic Pi

and Overtone [26], TidalCycles [27], ChuckK [28] and Extempore [29]. As long-time participants in the live coding scenes, we have observed a gradual shift from the experimentation of new languages and performance approaches towards a focus on musical expression. This could perhaps be understood as a sign of maturity of the field, when experimentation with the medium is gradually exhausted and the focus shifts onto the content. However, we believe that research in such a fruitful field as computer language design and human-computer interfaces for algorithmic composition is far from completed. There are multiple avenues of investigation lying before us.

The Sema project aims to re-inject the spirit of experimentation and linguistic plurality into live coding. We seek to introduce a new tool, a platform which enables live coders to create their own live coding languages and artificial neural networks, and to share them with other practitioners. The potential of machine learning as part of the live coding practice is yet to be explored, but can range from automation, to machine listening and interpretation through sound synthesis, to pattern generation and even code generation. We are looking forward to observing how live coders will appropriate, use and explore the Sema platform as a medium for live coding and how it can leverage new practices.

2 SEMA, A LIVE CODING LANGUAGE DESIGN PLAYGROUND

In 2018, we conducted a survey on language design with communities of practitioners of live coding, machine listening, machine learning and deep learning [30]. We asked participants which features they envisioned for future live coding environments and languages that could integrate machine listening and machine learning. The findings indicated a wide space of possibilities. Respondents suggested features such as support for hybrid approaches and multi-paradigm languages (i.e. object-oriented and functional), flexible, expressive and extensible languages and prototyping environments, good quality documentation and examples, as well as a clear and informative error report system. In a nutshell: brevity, simplicity, expressivity, flexibility, adaptability and plurality.

These findings prompted us to build *Sema*³, an open-source, modern Web browser -based system (Figure 1) to support rapid prototyping of multiple mini-languages for live coding, enabling users to satisfy their idiosyncratic expressive preferences. It seemed wiser to make a system where the user creates or refines their favourite language, rather than trying to satisfy everyone with one general design. Considering also the history and tradition of live coders building their own systems [17], this decision would contribute to a plurality of systems in a field that is already brimming with inventive solutions.

Our design exploration is helping us to find and refine the constraints, meta constraints and features for our

²<https://github.com/micknoise/Maximilian>

³<https://github.com/mimic-sussex/sema>

Fig. 1: Diagram of the architecture of the *Sema* live coding language design system.

system to support live coding language design. In early design iterations with *Sema*, we sought insights from user feedback in surveys and hands-on workshops [31] with *Sema*. We found that, although live code language design is not a trivial endeavour for people without a computer science background, it is possible to make this process achievable, more accessible and less cumbersome to other practitioners, such as the live coding community. In the following subsections, we introduce the fundamentals of the *Sema* playground and the design features we are creating to improve the user experience of the live coding, language design and ML workflows.

2.1 A Highly-Decoupled Reactive Architecture

The user feedback we collected in earlier studies [31] also revealed a need for a flexible and integrated environment which could enable, instruct and provide feedback on the different steps of the live code, language design, and ML workflows. One early design decision was the integration of the language design and grammar specification workflow in *Sema*. This step, in which the user compiles a language parser from a grammar specification (detailed in Sections 2.3 and 3.4) was initially accomplished externally to *Sema*, using a manual compilation process followed by the resulting source code integration.

The decision to support a full language design workflow prompted us to re-design our architecture to become more modular and integrated. This entailed the integration of an online compilation process and more generally, a scale up the complexity of *Sema*'s architecture (Figure 1). *Sema* has been designed with a modular, highly-decoupled architecture, in which core blocks such as the audio engine, worker threads and presentation layer are self-contained components which communicate using a message-oriented

subsystem (i.e. a publish-subscribe model with a message protocol).

We decided to explore modern UI frameworks and opted for Svelte⁴, a compiler for creating modern reactive Web applications. Svelte is very compatible with our own development principles and needs, and facilitated the implementation of a very lean and efficient solution. For instance, it facilitated the implementation of modular component hierarchies, reactive behaviour across workflows, and centralised in-memory state management with synchronisation with the browser's local storage. It also facilitated secondary features such as code instrumentation for logging of user activities across *Sema*.g. from single keystrokes, to layout changes, through to user access and change of tutorials and future integration of web services.

Sema's user interface (Figure 2) provides the user with a dashboard of customisable layouts and a library of dynamic components (i.e. widgets such as code editors, visualizers, debugging tools). These elements support *Sema*'s main user interface panels: the *Playground* and *Tutorials*. In the *Playground* panel, the user can customise his work space and freely explore the live coding, language design and ML workflows. In the *Tutorials* panel, the user may engage in a guided tour with tutorials, structured with different levels of granularity and gradually increased challenge.

2.2 Responsive Code Editors and Widgets

In previous design iterations of *Sema*, we experimented with basic integrations of code editor components. Our decision to opt for CodeMirror⁵ considered multiple criteria, including component architecture, modes and

⁴<https://svelte.dev/>

⁵<https://codemirror.net>

Fig. 2: Annotated screen capture of Sema's playground GUI (a live instance of v0.5.0, hosted at <https://sema.codes>), depicting a customisable dashboard, populated with editors for a) live code, b) ML model and c) grammar specification, a d) graphical audio analyser, and debugging widgets for f) AST and e) grammar.

syntax highlighting, community adoption, maintenance and support, and ease of integration with *webpack*⁶. CodeMirror powers all the editor instances which can be created in Sema, including:

- live coding editor (Fig. 2a), which provides users with continuous evaluation, key bindings for manual evaluation, suspension and resume of the audio engine playback, code commenting, and text search functions.
- generic JS editor (Fig. 2b), which enables users to live code their ML model architecture, data sets, model training and evaluation routines, and bind data streams and buffers to and from other components in Sema. All this code is dynamically evaluated in the scope of a dedicated ML worker which streams back results asynchronously.
- grammar editor (Fig. 2c), which enables users to specify, inspect or edit the different components of the user-definable grammar specifications (i.e. lexer, grammar production rules and semantic actions).

2.3 Language Processor and Intermediate Language

In order to parse, validate and evaluate instances of a live coding language from textual input in the editors, we need to be able to define its formal grammar. We opted for *Nearley.js*, a toolkit and library which implements the Earley [32] algorithm and generates JS parsers from a formal grammar specification in the Backus-Naur Form (BNF). *Nearley* allows a broader set of grammars than parsing expression grammars, a formalism which has been previously used in live coding language design (e.g. *PEG.js* [11] and *Ohm-js*⁷), including ambiguous grammars with left-side recursion.

The trade-off for the versatility and flexibility of *Nearley* is performance. Even with the integration of a tokeniser for higher parsing performance, benchmarks measured across browsers on a large JavaScript Object Notation (JSON) file,⁸ show that *Nearley* scores below domain-specific language (DSL) and hand-written parser implementations, parsing libraries, and other parser generators. Nevertheless,

⁷<https://github.com/harc/ohm>

⁸<https://sap.github.io/chevrotain/performance/>

⁶<https://webpack.js.org/>

preliminary tests that we present later in this paper show that its performance is reasonable for the latency requirements of live coding in which, typically, small files and code snippets are evaluated.

2.4 Machine Learning and Machine Listening Workers

Machine learning (ML) and musical machine listening⁹ (MML) are first-class citizens and core components of the Sema system. Typically, both ML and MML processes have computationally intensive stages that are better suited for execution on a dedicated thread, or eventually, in an external process. Even with interactive machine learning workflows [3], where end-users build custom ML models from small, lightweight, user-created data sets, when the main JS thread freezes during the ML model-training stage, it causes critical usability issues and undermines the user experience of any application [33]. As such, ML and MML processes in our system follow a loosely coupled architecture based on JS workers¹⁰: threads which leverage multi-core CPUs and communicate using asynchronous message exchange and event streams. These processes also adhere to our transducer concept, in that the sample rates from the event streams they generate are converted to and from the sample rate of the audio context.

The machine learning worker is populated with two complementary machine learning libraries. *TensorFlow.js* covers the majority of machine learning needs, and enables GPU processing. *MLweb* [34] provides further ML models, and includes the *LALolib* scientific computing library, which provides linear algebra functions that are not present in *Tensorflow.js*. Sema includes utility functions to translate matrices across the two libraries.

2.5 Design Requirements for the Signal Engine

In order to support a broad spectrum of live coding languages that ranges from abstract high-level languages to more powerful low-level languages, we strive for an adequate compromise between simplicity and flexibility. The following are the design principles that guided the implementation of our signal engine:

Integrated signal engine There is no conceptual split between the intermediate language and signal engine. Everything is a signal.

Single-sample signal processing Per-sample sound processing affords a broad set of sound control and synthesis processes, including more sophisticated and expressive techniques that use feedback loops, such as physical modelling, reverberation and infinite impulse response filtering [35].

Sample rate transduction It is simpler to do signal processing with one principal sample rate, i.e. the audio rate. Different sample rate requirements of dependent objects can be resolved by upsampling

and downsampling, using a ‘transducer’. The ‘transducer’ analogy enables us to accommodate a variety of processes with varying sample rates (e.g. video, spectral rate, sensors, ML model inference) within a single engine.

Minimal abstractions There are no high-level abstractions such as buses, synths, nodes, servers, or any language scaffolding in our signal engine. Such abstractions sit within the end-user mini-language design space.

We prioritise usability and learnability in the design of our signal engine and intermediate language. As such, some of these design principles prioritise usability over efficiency (e.g. single-sample processing [28]). Nevertheless, we are balancing out any performance trade-offs with a highly efficient implementation (as demonstrated in section 4).

In the next section we describe the design process and solution that these principles guided.

3 DESIGN STRATEGY

At the core of Sema’s signal engine is the Maximilian DSP library [15]. It is a free, open source and MIT-licensed audio synthesis and signal processing library implemented in C++. Maximilian was designed as a cross-platform and easy-to-use library for artists and creative computing students, and for rapid prototyping and commercial software development of interactive audio applications. The library provides a high-level interface to a comprehensive set of DSP primitives, including oscillators with standard waveforms, envelopes, filters with resonance, sample playback, delay-based effects, Fast Fourier Transform (FFT), granular synthesis, feature extraction, and multi-channel support.

The following sections present the elements of the design strategy we have adopted for the development of Sema’s signal engine and architecture. This includes rationale for the refactoring of Maximilian and creating a new Wasm build to execute on the AudioWorkletProcessor (AWP) scope, for the development of a compiler for DSP code, and mechanisms for code injection, dynamic evaluation and hot-swapping in this scope. Finally, this section discusses how of signal engine integrates with machine learning workers.

3.1 Refactoring the Maximilian Library for AudioWorklet

Previously, Maximilian was transpiled to an asm.js library using Emscripten [36].¹¹ The output consisted of single file with asm.js appended with custom JSe.g. a pre-assembled WAAPI graph with a user-definable closure injected at the audio callback of a SPN, JS TypedArray conversion to emscripten native types and asynchronous sample loading. This library provided a simple framework and abstraction over the WAAPI graphs

⁹<https://mimicproject.com/guides/mml1>

¹⁰<https://w3c.github.io/workers/>

¹¹<https://gitlab.doc.gold.ac.uk/mick/maxi-js-emscripten>

and nodes. Most usefully, the closure bound to the scope of the SPN audio callback provided users with a sandbox for accessing the audio buffer directly and implementing custom signal processing for interactive audio applications with Maximilian DSP classes.

The previous `asm.js` version of Maximilian revealed useful to end-users, particularly for teaching students about DSP and audio with the accessibility and convenience of a Web environment [15]. However, it suffered from the well-known limitations of SPN [10]. In earlier studies [33], we assessed users integrating Maximilian with machine learning and different data sources for rapid prototyping interactive audio applications. The main thread interrupts at the ML-training stage resulted in audio glitches and freezes that frustrated users. These usability issues were considered critical and would clearly undermine the adoption of these tools for more serious applications. This includes live coding performances where the quality of the audio is fundamental.

In order to overcome the SPN issues, and to arrive at a more scalable, efficient and usable design for a signal engine, we decided to advance towards an *AudioWorklet*-based implementation, and to refactor Maximilian.js accordingly.

3.2 WebAssembly generation with Emscripten

Refactoring Maximilian.js for *AudioWorklet* implied changing the build process to generate Wasm instead of `asm.js`. Compiling Maximilian to Wasm caused a breaking change that rendered the Maximilian DSP types unavailable in the SPN callback at load time, due to the asynchronous Wasm module instantiation. To ensure retro-compatibility to Maximilian.js SPN users, we restructured the build process to provide conditional builds of the library. The SPN version is modularised, exports with library name, and provides embindings with smart-pointer constructors for automated memory management and garbage collection of the DSP types. The Wasm version of the module was trimmed down and now only contains the embindings for loading the DSP classes in the AWP scope with normal constructors for finer control over object disposal. Listing 1 shows an excerpt of the embinding for *maxiOsc*, Maximilian's native type for standard wavetable oscillators.

Listing 1: Excerpt of Maximilian *maxiOsc* embinding

```
class_ <maxiOsc>("maxiOsc")
#ifdef SPN
  .smart_ptr_constructor(
    "shared_ptr<maxiOsc>",
    &std::make_shared<maxiOsc>)
#else
  .constructor <>()
#endif
  .function("sinewave", &maxiOsc::sinewave)
  .function("saw", &maxiOsc::saw)
```

The design pattern¹² for *AudioWorklet* with WebAssembly provides a restricted set of Emscripten compilation flags. We found that some of the previously used compilation flags, such as `MODULARIZE` prevented the Wasm module to load successfully and were removed.

```
CFLAGS=--bind -O3\
-s WASM=1 \
-s BINARYEN_ASYNC_COMPILATION=0 \
-s SINGLE_FILE=1 \
-s ALLOW_MEMORY_GROWTH=1 \
-s ABORTING_MALLOC=0 \
-s TOTAL_MEMORY=128Mb -DVORBIS \
-s "EXPORT_NAME='Maximilian'" \
-s "BINARYEN_METHOD='native-wasm'" \
--js-opts 0 -g4
```

To support the strategy of dynamic code evaluation, we found that it was necessary to add a compilation option to control the Wasm heap memory expansion. We will discuss this further in Section 5.

3.3 Audio worklet code injection

Upon the introduction of *AudioWorklet*, a few community examples emerged that used dynamic code injection (e.g. `dsp.audio's Worklet Editor`¹³ and A. Carabott's `Live Coding Playground`¹⁴). These examples de-serialise worklet processor templates in vanilla JS using `Blob`, manipulate them, and hot-swap them from memory into the worklet as interchangeable modules. Our first attempts explored this pattern using code generation and the additional constraint of dynamically loading the Maximilian Wasm module into the AWP scope. This constraint introduced a series of difficulties revealed by the "*Error on loading worklet: DOMException*" error message. The ambiguity of the error message and the constraints of the AWP scope caused a challenging debugging process.

To solve this problem, we considered different tactics. We found out that the AWP scope excluded some of the APIs that enable the first alternatives. We also found out through `StackOverflow`¹⁵ that the opaque origin of the worklet processor requires absolute URIs to load the module asynchronously. This unveiled another problem related to the unsuccessful parsing of the Wasm module when it was imported by processor code that was generated and loaded dynamically from a `Blob`. Eventually, we turned to a more creative solution.

¹²<https://developers.google.com/web/updates/2018/06/audio-worklet-design-pattern>

¹³<http://dsp.audio/editor>

¹⁴<https://acarabott.github.io/audio-worklet-live-coding>

¹⁵<http://bit.ly/2F4vZKM>

3.4 Parsing, evaluation and execution of user-defined code

In this section, we present the solution that we have arrived at for supporting the evaluation of the DSP code compiled from user-defined live code. Fundamentally, it consists in utilising the worklet processor as a template that imports the Maximilian Wasm module and loads its DSP classes into the AWP scope. The AWP is where the DSP code is injected and dynamically evaluated.

When users evaluate an expression in the Sema live coding editor, i.e. by pressing Cmd-Enter after selecting an expression or placing the cursor on a given line they are triggering a specific workflow in our system (Figure 1). First, the user-evaluated expression is parsed by the Nearley-generated parser. If the expression is valid according to the language formally defined by the BNF grammar specification, the parser outputs an Abstract Syntax Tree (AST), a tree-like data structure that breaks down the user expression. The AST is converted into serialised JS expressions that specify which Maximilian DSP objects are used and how they are assembled into DSP functions that will run on the main audio loop `setupFunction` and `loopFunction`, respectively. These JS expressions are packed into a JS object which is posted through the processor *AudioWorkletNode* messaging port (Listing 2).

Listing 2: Code injection into the AWP scope

```
this.audioWorkletNode.port.postMessage({
  eval: 1,
  setup: userInterpretedFunction.setup,
  loop: userInterpretedFunction.loop
});
```

Once the message containing a payload of serialised DSP JS resolves asynchronously in AWP scope, the code is evaluated just-in-time (JIT) using the `eval()` function. The user-evaluated expression that was converted to a JS DSP specification with Maximilian objects is compiled and set ready to run in the AWP loop (Listing 3).

Listing 3: Dynamic code evaluation in the AWP scope

```
this.port.onmessage = event => {
  if ("eval" in event.data) {
    try {
      let setupFunction =
        eval(event.data["setup"]);
      let loopFunction =
        eval(event.data["loop"]);
      ...
    } catch (err) {
    }
  }
}
```

For instance, considering the following expression for FM synthesis (Listing 4), that is valid for of a live coding language that a user has defined through a given grammar:

Listing 4: Example of a test user-evaluated DSP expression

```
osc sin(osc sin 100 + osc tri 40)
```

The system interprets this expression and generates two anonymous functions. The *setupFunction* (Listing 5) declares and instantiates the required Maximilian Wasm objects for the custom DSP expression in the AWP scope.

Listing 5: The *setupFunction* generated from the AST

```
() => {
  let q=[];
  q.osc13 = new Maximilian.maxiOsc();
  q.osc14 = new Maximilian.maxiOsc();
  q.osc15 = new Maximilian.maxiOsc();
  return q;
}
```

The original user-expression is converted to a compiled JS function, the *loopFunction* (Listing 6), which composes the Maximilian objects previously declared in *setupFunction* and runs in the AWP *process* loop (the main audio loop). The audio engine crossfades between the previous *loopFunction* and the newly generated one.

Listing 6: The *loopFunction* generated from the AST

```
(q) => {
  return q.osc13.sinewave((
    q.osc14.sinewave(100)
    + q.osc15.triangle(40))
  );
}
```

3.5 Connecting Sound and Machine Learning

As detailed earlier, machine learning and audio processing are carried out in two separate threads; audio in an audioworklet, and machine learning in a webworker. This enables the two threads to run on separate CPUs without interfering with each other's performance. However the two threads still need to be connected to share data streams. An example use case could be a machine learning model that carries out a transformation on spectral data; to enable this, the output from a Fast Fourier Transform (FFT) would be sent from the audio engine to the model, and the output from the model would be sent back to the audio engine and processed with an Inverse FFT.

Sema has two functions, *toJS* and *fromJS* to enable this routing between audio and machine learning. *JS* in the names indicates the *JavaScript* window. On the Javascript side, the user sets up a callback function *input* which has an identity number and a data item as parameters. On the audio side, any data streamed to the *toJS* function will be passed to the callback via internal messaging. *toJS* is specified with a polling frequencies that controls the rate at which data is sampled. The reverse process happens with the *output* function on the JavaScript side, which is received by the *fromJS* function. The audio engine functions are part of the intermediate language specification. They can be used on their own, or wrapped up into larger language constructs so that audio-machine learning connections are transparent to the user.

The audio and machine learning domains are further linked by signal buffers. Using the function *sema.sendBuffer*, the user can generate and send a named buffer to the audio engine, which can be used with sample playback functions. This feature allows the audio engine to use non-realtime generated content from machine learning models.

4 PERFORMANCE AND RELIABILITY EVALUATION

Good performance and reliability are central measures of the success of a live coding system, which must be able to meet the demands of live musical performance. We measured the reliability, latency and signal processing capacity of our system. To achieve this, we designed a simple testing language that was capable of creating nested trees of oscillators. An algorithm was designed to probabilistically generate code for oscillator trees with variable width and depth. An example of generated code is shown below.

```
osc sin (osc sin 960 + osc sin 961 +
osc sin (osc sin 657 + osc sin 348 +
osc sin 405) + osc sin (osc sin 1050 +
osc sin 122 + osc sin (osc sin 1052 +
osc sin 1090 + osc sin (osc sin 1088 +
osc sin (osc sin 258) +
osc sin (osc sin (osc sin 820))) +
osc sin 221)))
```

This code generator could be hand tuned to change the probable size of its output. The test procedure worked as follows: (1) generate some code, (2) record the number of oscillator objects in the code, (3) run the code in the audio engine, recording times for parsing, JS code generation from the parse tree, and for compiling the code using *eval()*, (4) wait for 200ms and repeat.

All tests run in Chrome 75.0, on a MacBook Pro mid-2015, with 16Gb RAM and a 2.5GHz i7 CPU¹⁶. We recorded time using *window.performance.now()*, with sub-millisecond resolution. These results figures measure the capacity of a single audio worklet, rather than the capacity of the browser's JS virtual machine, which is capable of running multiple audio worklets.

4.1 Signal Processing Capacity

Our first investigation tested the signal processing limits of the system, in a test designed to mimic real-world scenarios. The code generator was configured to create code that might overload the AWP signal processing loop. When overloading happens, this has an effect on the *eval()* time of the subsequent test, because the *eval()* command is carried out in the AWP thread and slows down if it is overloaded. We ran the test for one hour, over 14000 iterations. Figure 3 shows a scatter plot comparing the

number of sine wave oscillators running in the previous test compared to the time taken for *eval()* to run in the current test. The *eval()* time is stable until around 200 oscillators, when variance starts to increase, and then above 250 oscillators we start to see an increase in latency. Parsing and code generation times remain stable throughout, as they are unaffected, running in a separate worker thread. From these results, we can estimate that the system will run with good stability with approximately 200-250 sine wave oscillators on the test machine.

4.2 Reliability and Latency

We tested the latency of the system within the stable limits discovered in the previous test. This is the time between when a user evaluates a line of code, and when the compiled code is ready to run in the next DSP cycle. The test ran for ten hours and 11 minutes, with 173422 iterations. Extreme outliers (with z-score over 20) were removed, accounting for 0.007% of datapoints. The remaining 99.993% of tests show latency of <25ms, indicating high reliability. Figure 4 shows individual latencies compared to the number of sine wave oscillators in the test code. The largest and most variable cause of latency is the Nearley parser. The combined latency times (Figure 5) demonstrate that the system is likely to run with sub-perceptual (<20ms) latencies when running within stable limits. The blue lines in both figures show a linear regression fit.

Fig. 3: Load testing results

Fig. 4: Latency test results: individual processes

¹⁶CPU Model with "sysctl -n machdep.cpu.brand_string" on MacOS terminal: i7-4870HQ CPU @ 2.5 GHz

5 DISCUSSION AND DESIGN PATTERNS

The JS community often considers the use of the `eval()` function a bad coding practice [37]. However, it has been used in the Webkit console, JSBin, and the widely adopted library `lodash`¹⁷. The problems of using `eval()` have been associated with malicious attacks through code injection and cross-site scripting, hard maintenance and debugging, slow performance due to the cost of compilation, and wastefulness of resources (e.g. creating execution context collecting garbage, exporting variables). While we are still exploring this solution within the constraints of our scenario e.g. client-side, immediate execution in the AWP scope our evaluation metrics suggest that the advantages of `eval()` supersede the potential problems. One problem that we found with using `eval()` and Wasm was in the heap memory expansion. Although Emscripten provides an `ALLOW_MEMORY_GROWTH` compilation option, we found it led to inconsistent behaviours. We resolved this issue by setting a maximum memory growth threshold.

In the exploration and gradual development of our solution, we dealt with difficulties arising from the integration of technologies with different degrees of maturity. Some difficulties inherent to *AudioWorklet* were, for instance, error messaging, the availability of APIs in the AWP scope, and its impact in the Wasm module loading (e.g. for standard modularisation and module naming). Other difficulties emerged from loading these components within a webpack development environment which revealed somewhat cumbersome (e.g. adding imports for emitting Wasm without compilation, server MIME type definitions).

Our solution can provide a design pattern for people interested in re-using our stack or developing a similar approach. The first use of the signal engine was in a live coding performance at AlgoMech 2019 by MNISTREL¹⁸. The artist integrated the signal engine in his own personal web site and live coded directly in Chrome Dev tools console, making it his main live coding environment. The signal engine has since been used in multiple

¹⁷Angus Croll: Break all the rules, JSConf 2012, <https://youtu.be/MFtijdk1ZDo>

¹⁸MNISTREL at Algo/Mesh 2019, <https://mnistrel.com/algomesh2019>

Fig. 5: Latency test results: combined processes

live performances with Sema, including an evening of performances by artists that attended a week-long workshop on the Sema system [38], and by the authors at the Web Audio Conference 2019 and ICLC 2020.

6 CONCLUSION

We present a high-performance implementation of a signal engine for a new browser-based system for live coding. On its own, Sema's signal engine is a piece of infrastructural open-source software, which can be extracted and reused by other Web-based applications. It can be useful as an alternative solution for a high-performance engine and can therefore contribute to open up the benefits of client-side Web applications for sound and music (e.g. zero-installation, global accessibility and distribution). We hope that the documentation of our exploration process and solution can distil a useful design pattern and help, not only the live coding and web audio hacker communities, but the wider music tech community to tap into these benefits.

Sema, the system we designed for empowering users to design and perform with their own live coding languages and machine learning, determined the design constraints and decisions of our signal engine implementation. In Sema, user-designed languages are mapped onto an intermediate language, which is compiled to Maximilian JS DSP code. This code is injected, dynamically evaluated and hot-swapped in the *AudioWorkletProcessor* scope of the signal engine. The signal engine also affords real-time communication with Sema's machine learning subsystems, which are also livecoded in JavaScript. Evaluation shows that the system ran with high reliability on the test machine for 200-250 oscillators, with sub-perceptual latency. Most notably, the new *AudioWorklet* implementation allowed us circumvent the well-established limitations from the *ScriptProcessorNode* and reuse the Maximilian code base to fit the requirements of musical live coding performance.

Future work will focus on the presentation layer, user language specification support, instructional design and further development of the ML and MML workers. These developments include increasing the compilation complexity to support more powerful and expressive user-defined languages, optimising JIT compilation for parser and DSP code generation, further optimisation of the Maximilian library codebase, and supporting for inter-thread communication with *SharedArrayBuffer*.

7 ACKNOWLEDGMENTS

We would like to thank the participants in our Sema workshops for their participation and contributions to the open-source repository.

This work has received funding from UK Arts and Humanities Research Council for the MIMIC project (ref AH/R002657/1, <https://mimicproject.com>).

8 REFERENCES

- [1] W. K. Edwards, V. Bellotti, A. K. Dey, M. W. Newman, "Stuck in the Middle: The Challenges of User-Centered Design and Evaluation for Infrastructure," presented at the *Proceedings of the SIGCHI conference on Human factors in computing systems - CHI '03*, pp. 297–304 (2003), <https://doi.org/10.1145/642611.642664>.
- [2] M. Fowler, "Module Assembly," *IEEE SOFTWARE*, pp. 65–67 (2004).
- [3] F. Bernardo, M. Zbyszyski, M. Grierson, R. Fiebrink, "Designing and Evaluating the Usability of a Machine Learning API for Rapid Prototyping Music Technology," *Frontiers in Artificial Intelligence*, vol. 3, p. 13 (2020), <https://doi.org/10.3389/frai.2020.00013>.
- [4] W3C Audio Working Group, "Web Audio API," <https://webaudio.github.io/web-audio-api> (2019).
- [5] Y. Mann, "Interactive Music with Tone.js," *Proc. of the 1st annual Web Audio Conference* (2015).
- [6] C. Roberts, "Metaprogramming Strategies for AudioWorklets," presented at the *Web Audio Conference* (2018).
- [7] M. Grierson, C. Kiefer, "Maximilian: An Easy to Use, Cross Platform C++ Toolkit for Interactive Audio and Synthesis Applications," presented at the *Proc. of The International Computer Music Conference*, August, pp. 276–279 (2011).
- [8] S. Letz, S. Denoux, Y. Orlarey, D. Fober, "Faust audio DSP language in the Web," *Linux Audio Conference*, pp. 29–36 (2015).
- [9] V. Lazzarini, E. Costello, S. Yi, "Csound on the web," presented at the *Linux Audio Developers Conference* (2014), URL <http://opus.bath.ac.uk/39719/>.
- [10] H. Choi, "AudioWorklet: The future of web audio," presented at the *Web Audio Conference* (2018).
- [11] G. Wakefield, C. Roberts, "A Virtual Machine for Live Coding Language Design," *Proc. of New Interfaces for Musical Expression 2017*, pp. 275–278 (2017).
- [12] D. Smilkov, N. Thorat, Y. Assogba, A. Yuan, N. Kreeger, P. Yu, K. Zhang, S. Cai, E. Nielsen, D. Soergel, S. Bileschi, M. Terry, C. Nicholson, S. N. Gupta, S. Sirajuddin, D. Sculley, R. Monga, G. Corrado, F. B. Viégas, M. Wattenberg, "TensorFlow.js: Machine Learning for the Web and Beyond," presented at the *Proc. of the 2nd SysML Conference* (2019), URL <http://arxiv.org/abs/1901.05350>.
- [13] A. Roberts, C. Hawthorne, I. Simon, "Magenta.js: A JavaScript API for Augmenting Creativity with Deep Learning," presented at the *Proc. of the 35th International Conference on Machine Learning*, pp. 2–4 (2018).
- [14] D. Shiffman, J. Ashe, K. Compton, H. Davis, D. Kazemi, G. Kogan, K. McDonald, Yining Shi, "ml5: Friendly Open Source Machine Learning Library for the Web," *ADJACENT*, , no. 3 (2018).
- [15] M. Grierson, M. Yee-king, L. McCallum, C. Kiefer, M. Zbyszyski, "Contemporary Machine Learning for Audio and Music Generation on the Web: Current Challenges and Potential Solutions," *ICMC* (2018).
- [16] TOPLAP, "Manifesto," <https://toplap.org/wiki/ManifestoDraft> (2010).
- [17] T. Magnusson, "Herding Cats: Observing Live Coding in the Wild," *Computer Music Journal*, vol. 38, no. 1, pp. 91–101 (2014), [10.1162/COMJ.2014.01.00001](https://doi.org/10.1162/COMJ.2014.01.00001).
- [18] C. Roberts, J. Allison, D. Holmes, B. Taylor, M. Wright, J. Kuchera-Morin, "Educational design of live coding environments for the browser," *Journal of Music, Technology and Education*, vol. 9, no. 1, pp. 95–116 (2016), https://doi.org/10.1386/jmte.9.1.95_1.
- [19] M. Yee-King, M. Grierson, M. D'Inverno, "STEAM WORKS: Student coders experiment more and experimenters gain higher grades," *IEEE Global Engineering Education Conference, EDUCON*, , no. April, pp. 359–366 (2017), <https://doi.org/10.1109/EDUCON.2017.7942873>.
- [20] N. Collins, A. McLean, J. Rohrerhuber, A. Ward, "Live coding in laptop performance," *Organised Sound*, vol. 8, no. 3, p. 321330 (2003), [10.1017/S135577180300030X](https://doi.org/10.1017/S135577180300030X).
- [21] A. McLean, "Hacking Perl in Nightclubs," (2004).
- [22] J. Armitage, "Spaces To Fail In: Negotiating Gender, Community and Technology in Algorave," *Dancecult: Journal of Electronic Dance Music Culture*, vol. 10, no. 1, pp. 31–45 (2018).
- [23] J. McCartney, "Rethinking the computer music language: SuperCollider," *Computer Music Journal*, vol. 26, no. 4, pp. 61–68 (2002), <https://doi.org/10.1162/014892602320991383>.
- [24] T. Magnusson, "The IXI Lang: A Supercollider Parasite For Live Coding," presented at the *Proc. of the International Computer Music Conference*, pp. 5–8 (2011).
- [25] C. Roberts, J. A. Kuchera-Morin, "Gibber: Live coding audio in the browser," *ICMC 2012: Non-Cochlear Sound - Proceedings of the International Computer Music Conference 2012*, , no. January, pp. 64–69 (2012).
- [26] S. Aaron, A. F. Blackwell, "From Sonic Pi to Overtone: Creative Musical Experiences with Domain-specific and Functional Languages," presented at the *Proceedings of the First ACM SIGPLAN Workshop on Functional Art, Music, Modeling FARM'13*, pp. 35–46 (2013), <https://doi.org/10.1145/2505341.2505346>.
- [27] A. McLean, "Making Programming Languages to Dance to: Live Coding with Tidal," presented at the *Proceedings of the First ACM SIGPLAN Workshop on Functional Art, Music, Modeling FARM'14*, pp. 63–70 (2014), <https://doi.org/10.1145/2633638.2633647>.
- [28] G. Wang, P. R. Cook, S. Salazar, "ChucK: A Strongly Timed Computer Music Language," *Computer Music Journal*, vol. 39, no. 4, pp. 91–101 (2015), <https://doi.org/10.1162/COMJ.2015.01.00001>.
- [29] A. C. Sorensen, *Extempore: The design, implementation and application of a cyber-physical programming language*, Ph.D. thesis, Australian National

University (2018), URL <http://hdl.handle.net/1885/144603>.

[30] C. Kiefer, T. Magnusson, “Live Coding Machine Learning and Machine Listening: A Survey on the Design of Languages and Environments for Live Coding,” presented at the *Proc. of the International Conference on Live Coding*. (2019).

[31] F. Bernardo, C. Kiefer, T. Magnusson, “Designing for an Accessible and Pluralist Live Code Language Ecosystem with Sema,” presented at the *Proc. of the 6th International Conference on Live Coding* (2020).

[32] J. Earley, “An efficient context-free parsing algorithm,” *Communications of the ACM*, vol. 13, no. 2, pp. 94–102 (1970), <https://doi.org/10.1145/362007.362035>.

[33] F. Bernardo, M. Grierson, R. Fiebrink, “User-Centred Design Actions for Lightweight Evaluation of an Interactive Machine Learning Toolkit,” *Journal of Science and Technology of the Arts*, vol. 10, no. 2, p. 2 (2018), <https://doi.org/10.7559/citarj.v10i2.509>.

[34] F. Lauer, “MLweb: A toolkit for machine learning on the web,” *Neurocomputing*, vol. 282, pp. 74–77 (2018).

[35] K. Steiglitz, *A Digital Signal Processing Primer: With Applications to Digital Audio and Computer Music* (Addison-Wesley) (1996).

[36] A. Zakai, “Emscripten: An LLVM-to-JavaScript Compiler,” presented at the *Proc. of the ACM international conference companion on Object-Oriented Programming, Systems, Languages, and Applications* (2011).

[37] D. Crockford, *JavaScript: The Good Parts* (O’Reilly Media), first edit ed. (2008).

[38] F. Bernardo, C. Kiefer, T. Magnusson, “Designing for a Pluralist and User-Friendly Live Code Language Ecosystem with Sema,” presented at the *International Conference on Live Coding* (2019).

NOMENCLATURE

AST = Abstract Syntax Tree

AWP = Audio Worklet Processor

BNF = Backus-Naur Form

DSP = Digital Signal Processing

JS = JavaScript

JIT = Just-in-time

ML = Machine Learning

MML = Musical Machine Learning

MIME = Multipurpose Internet Mail Extensions

NIME = New Interfaces for Musical Expression

SPN = ScriptProcessorNode

WAAP = Web Audio API

Wasm = Web Assembly

THE AUTHORS

Francisco Bernardo

Chris Kiefer

Thor Magnusson

Francisco Bernardo is a computer scientist, an interactive media artist and a multi-instrumentalist. His work focuses on Human-centred design of toolkits to broaden and accelerate user innovation with interactive machine learning. He has been working at the interface between academia and industry, mostly on applied research in public-funded projects at the intersection of art and innovation, front-end software engineering, user experience and interaction design, and greenfield product management. Currently, he is an Associate Tutor at the University of Sussex and a Postdoctoral Research Fellow at project MIMIC, where he develops Web-based environments for live coding machine learning

and language design. He also performs as MNISTREL. Further information at: <https://fbernardo.com>.

Chris Kiefer is a computer-musician, musical instrument designer and Lecturer in Music Technology at the University of Sussex. He performs with custom-made instruments including malleable foam interfaces, interactive sculptures and hacked acoustic instruments. As a live-coder he performs under the name Luuma. Most recently he has been playing an augmented self-resonating cello as half of improv-duo Feedback Cell, and with feedback-drone-quartet Brain Dead Ensemble. His

research specialises in musician-computer interaction, physical computing, machine learning and complex systems, with an emphasis on nonlinear and dynamical systems.

Thor Magnusson is a Professor in Future Music at the University of Sussex. His work focuses on the impact digital technologies have on musical creativity and practice, explored through software development, composition and performance. He is the co-founder of ixi audio (www.ixi-audio.net), and has developed audio software, systems of generative music composition, written computer music tutorials and created two musical live coding environments. He has taught workshops

in creative music coding and sound installations, and given presentations, performances and visiting lectures at diverse art institutions, conservatories, and universities internationally. In 2019, Bloomsbury Academic published Magnussons monograph *Sonic Writing: The Technologies of Material, Symbolic and Signal Inscriptions*. The book explores how contemporary music technologies trace their ancestry to previous forms of instruments and media, including symbolic musical notation. The book underpins current research, where, as part of the MIMIC project (www.mimicproject.com), Magnusson has worked on a system that enables users to design their own live coding languages for machine learning. Further information here: <http://thormagnusson.github.io>