Making Computer Music on the Web with JSPatcher

Michel Buffa, Laurent Pottier, Shihong Ren, Yang Yu

Making Computer Music on the Web with JSPatcher

Shihong Ren
Shanghai Conservatory of Music, SKLMA, China
Université Jean Monnet, ECLLA, France
shihong.ren@univ-st-etienne.fr

Laurent Pottier
Université Jean Monnet, ECLLA, France
laurent.pottier@univ-st-etienne.fr

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

Yang Yu
Shanghai Conservatory of Music, SKLMA, China
yuyang@shcmusic.edu.cn

ABSTRACT

Web technology through the internet and a web browser, is attractive to modern computer music artists as it provides high accessibility and interactivity and opens to possibilities. Since the development of JSPatcher [1], an online visual programming language (VPL) for audio processing and interactive web-based programs, we have recently added various music-related features to the application, including important features that common music computing practices need, such as file management, audio buffer handling, audio plugin support, and computer-assisted composition functions. It aims to make it easier for musicians, composers or designers of multimedia projects who might be familiar with VPLs on native platforms like Max [2], PureData [3] or OpenMusic [4], to bring or create their work on the web with less effort. This paper presents the concept, implementation details and examples of these newly added features.

1. INTRODUCTION

JSPatcher is a web-based visual programming language (VPL) originally designed for providing a user interface (UI) for Web Audio API. [1] Since the API describes the audio processing flow as a graph of DSP nodes, it is convenient to have a *patcher* editing system [5] to manipulate the audio graph. JSPatcher is initially a WebAudio patcher editor that runs in a browser, where users can create boxes representing the DSP nodes and cables representing the connections in a canvas.

Since the whole JSPatcher platform is mainly developed in TypeScript and compiled to JavaScript which is the scripting language for the Web API, we have the possibility to fully import the language it self to the patcher system. Thus, in addition to the audio connection layer, in order to control DSP parameters with non-audio data, we added a dataflow layer that can distribute events between functions in real time. In this layer, various functions imported from the web environment, including JavaScript language built-ins and Web APIs, are available. These can be used to access the browser's high-level APIs, such as those for managing mouse and keyboard events, battery life or computer peripherals.

With these two layers in a single *imperative patcher*, the usage becomes similar to some VPLs available on native platforms like Max or PureData, while offering more flexible computational possibilities, taking advantages from the web community, as most of the JavaScript packages can be imported and used as functional boxes in a patcher.

An additional patcher interpretation mode, FAUST *compiled patcher*, has also been developed to facilitate DSP design under the WebAudio AudioWorklet specification [6]. The FAUST [7] language ecosystem is used to transform patchers under a specific mode into FAUST code which can be compiled to a customized DSP. This kind of can be used in imperative patchers as a special WebAudio node called AudioWorkletNode that can be connected with other WebAudio nodes. With this interpreter, some Max's Gen patchers can also be interpreted to FAUST code and be compiled in the same way in JSPatcher.

These new features make JSPatcher not only a utility to interface graphs from the Web Audio API, but also an integrated development environment (IDE) for data computation and audio processing on the web. Various possibilities related to music composition and performance are thus open for exploration on the platform.

The following sections will present the improvements made to JSPatcher that contain important features for modern computer music practices, including a file manager, audio buffer manipulation, audio plugin support and computer-assisted composition functions.

2. FILE MANAGER

2.1 Concept

One major limitation of the web environment is the access to the device's local files due to security reasons. According to the current web standards, web applications do not have permissions by default to read local files unless users explicitly allow them. The lack of a proper file manager API makes any web application difficult to maintain the structure of a file-based project.

To support sub-patchers and audio file manipulation in

[1] https://www.w3.org/TR/webaudio/

Figure 1. JSPatcher with a file manager

JSPatcher, we had to design a virtual file system in the browser that allows the following operations:

1. Make the file system persistent after closing the application,
2. Upload files from the user's machine,
3. Create new files,
4. Delete files,
5. Copy or move files,
6. Access file data using its path.

Designing and implementing a persistent file storage system is a challenge in this work as there are few ways to keep reusable data in the browser. Our solution relies on IndexedDB, [2] a technology for client-side storage of significant amounts of structured data. Normally, in 2022 and on a desktop browser, up to two gigabytes of storage quota are available in the IndexedDB per site group, [3] . It is sufficient for any lightweight project. We also use BrowserFS, a JavaScript module that can store persistently a whole structured file system into the IndexedDB and provide a JavaScript API for file management [8, 9].

To accelerate file reading and writing in runtime, we also built a higher-level file system that caches the file tree structure and data in memory. It acts as a bridge between the BrowserFS and the UI, when a change is made by the user to the file system, it calls the BrowserFS to store new change in IndexedDB; meanwhile, all event subscribers to the changed file, such as the file manager UI (Figure 1), file editors or audio players will be alerted and they can react in real time.

2.2 Temporary File System

Along with the persistent file system in IndexedDB, we also implemented another memory-only file system for temporary entries. It is internally a label and file content map with event emitters and an observation mechanism.

The purpose of the temporary file system is to allow users to create memory spaces and use them with an associated name just like a persistent project file. A similar behavior exists in Max. For example, when a user defines an audio buffer using a name that doesn't refer to a file from the hard disk, the system will allocate a part of the memory that is temporarily accessible with this name. Any object that uses the same name will then refer to the same part of the memory. When every object associated with this name is removed, the memory will be freed and erased.

In JSPatcher, this behavior is implemented by the temporary file system in which every file is under the root directory. We have provided an API for objects to create temporary files when a name cannot be resolved as a persistent file. Meanwhile, the object becomes an observer of the temporary file. When the last observer gets removed, the temporary file will automatically free the memory space and remove itself.

2.3 File Structure of a Project

In JSPatcher, a project is a folder that contains subfolders or files in any format as in native operating systems. When a session is started, the project folder will automatically be initialized with a hidden file named `.jspatproj` under the root folder. This file contains metadata related to the project such as its name, its author, its dependencies and will be saved with the whole project.

Typically, a project will have one or multiple patchers with some asset files. To distinguish patchers under different modes, we use `.jspat` as the extension for regular imperative patcher, and `.dsppat` for FAUST compiled patcher. Patcher files can be uploaded to the file system or downloaded (exported) from it. Users can also upload or download the whole project for further exchange under the `.zip` format with every file compressed inside.

2.4 Usage in Patchers

Some types of project files can be recognized by JSPatcher and can be loaded into patchers. Typically, they will be used in a JavaScript patcher (using imperative and WebAudio layers) by different box objects. To load a specific file, the user should put the file path in Unix format (using slash for sub-directory) relative to the project root folder.

Table 1 shows a list of file types that can be loaded by box objects in a patcher. Figure 2 shows a sub-patcher file, an audio file, a text file and an image file in a patcher.

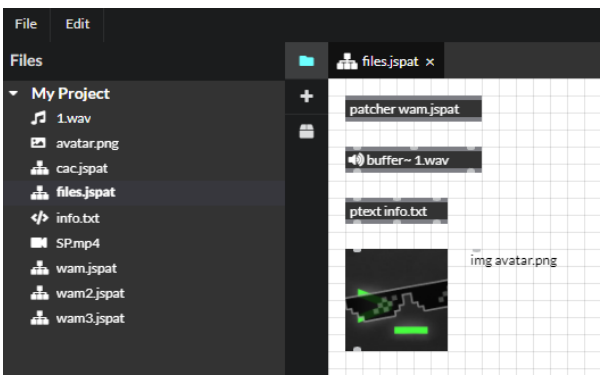

Figure 2. Files in a patcher

[2] `https://www.w3.org/TR/IndexedDB/`
[3] `https://tinyurl.com/muc8cmyx`

Extension	Loader	Description
`jspat`	`patcher`	Sub-patcher
`dsppat`	`pfaust`	Faust sub-patcher that can be compiled to an AudioWorklet DSP
`gendsp`	`gen`	Gen sub-patcher that can be compiled to an AudioWorklet DSP
Audio files (`wav`, `aif`, `mp3`, etc.)	`buffer~`	Audio files that can be decoded to in-memory audio buffers for further manipulations
Image files (`jpg`, `png`, `gif`, etc.)	`img`	Display the image in the patcher
Text files (`txt`, `json`, etc.)	`ptext`	Edit the text in the patcher

Table 1. A list of file types that can be loaded by box objects in a patcher.

3. AUDIO EDITING

3.1 Concept

Different kinds of web-based audio editors or digital audio workstations (DAW) have been developed by the community in recent years to bring desktop audio production experience on the web platform. However, for a long period, due to the inconsistent implementation of the Web Audio API between different web browsers, it has been hard to maintain related projects and ensure the same user experience across browsers and devices.

Support for the AudioWorklet API on Safari Desktop and iOS browsers was added in April 2021, which as the final step in providing full Web Audio API coverage on all major browsers. Also, the Web Audio API version 1.0 became, after ten years of maturation, a W3C recommendation (a frozen standard) in June 2021. These two events mark a milestone of the standard and its implementation process. It motivated developers to adapt Web Audio applications to the recommended standard without worrying about compatibility issues. The AudioWorklet API is important for audio editors or DAWs as buffer recording, looping, editing regions require sample-accurate controls over the input signal and can be done in a customized DSP through the API.

In JSPatcher, using the recent API, two features related to audio files were added. Audio files can now be edited in non-real time from a single-track audio editor and can be manipulated as an audio buffer in real time in a patcher.

3.2 Audio Editor

In the workflow for some genres of electronic music, especially *musique concrète* or interactive electroacoustic music, raw audio materials need to be cut and "cleaned up" before being put into a DAW project or a program. This process requires a tool that can directly modify the audio file with the following features:

1. Visualize any part of the waveform with rulers,
2. Play any part of the audio clip,
3. Select on the waveform with sample-accuracy,
4. Record audio in-place or after the clip,
5. Cut, copy and paste any part of the clip,
6. Adjust the volume of the selected section,
7. Silent the selected section or insert silence of any length,
8. Perform a phase inversion or reverse the selected section,
9. Fade-in and fade-out of any length with a flexible curve,
10. Resample the audio to any sample rate,
11. Create, delete, reorder channels, up-mix and down-mix (i.e., mix a stereo audio to a mono one),
12. Apply audio effects,
13. Export the audio in some formats.

Its implementation in JSPatcher is a dedicated window that can be opened by double-clicking an audio file listed in the file manager. (Figure 1)

When the user needs to open the editor, the system will firstly decode the given file to raw PCM data. Then, the data will be grouped into different levels of zoom that contain minimum and maximum values of each 16, 256, 4096 samples, etc. to optimize the waveform display for long audio files.

The layout of the editor's UI is similar to some desktop audio editors. A navigation bar with the waveform of the whole file is shown at the top of the window with a range of currently displayed and selected sections. Below the bar, a larger waveform viewer shows a section of the audio. Conventional features were implemented such as the cursor, the auto-scalable timing grid (vertical), energy (in dB) grid (horizontal), buttons to enable or disable channels for replay, fade-in and fade-out handlers, and an additional popup handler to adjust the gain when a section is selected.

Below the main viewer, a playback toolbar is presented. Users can play, pause or stop the playback from the cursor's position. Loop can also be enabled or disabled here. To record audio, users can choose an input device from a menu, enable the monitoring, and start recording using the red button.

At the bottom, a meter shows the current volume of the playing or the monitoring audio. A table with adjustable numbers shows the current displayed and selected section, and allows users to change it manually.

In the menu, a set of options are available to perform simple processing like silence, reverse, phase inversion, insert silence, resample, remix channels and export. To perform more complex processing using third-party DSPs, it is possible to load WebAudioModule (WAM) plugins [10, 11] into a plugin rack. Users can choose to process the whole audio file or only the selected part. We will present the WAM support in the next sections of the paper.

In the "Remix channels" option, users have access in a popup window to an I/O matrix to decide the number of outputs and how much volume of signals coming from

original channels for the resulting audio. (Figure 3)

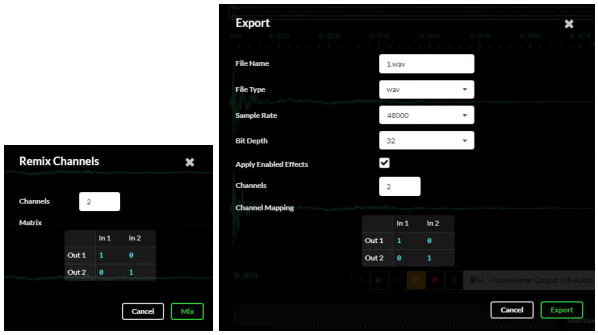

Figure 3. "Remix channels" and "Bounce" options

The audio export ("Bounce" option) supports `.wav`, `.mp3` and `.aac` formats with different sample rates or bit-rates thanks to a web implementation of ffmpeg [4] through Emscripten [5] compilation and WebAssembly. [6] [7] (Figure 3)

3.3 Audio Buffer Manipulation in a Patcher

Audio buffer manipulation is a key feature for any audio programming environment like Max or JSPatcher, as real time audio projects like some interactive music pieces often need to record and replay audio clips during the performance.

In the previous section, we mentioned that the `buffer~` object can refer to any audio file in the project or can create any temporary audio clip in the memory. The audio editor is also available when the user double-clicks the object. In addition, the editor can be "docked" beside the patcher.

The `buffer~` object accepts 4 possible arguments. The first argument is the identifier of the audio file or a temporary audio clip. If it is temporary, users can initialize it by specifying the buffer's number of channels, length in samples and its sample rate.

In fact, the object is associated with a PatcherAudio API in which the audio buffer is stored with its waveform data, with a set of methods related to the buffer editing. A Bang (a triggering event) from the first inlet of the `buffer~` object will trigger output of the PatcherAudio API.

Users can connect the `buffer~` object to some other objects to use the PatcherAudio API. For example, the `waveform~` object can display the waveform of the audio clip, `bufferSource~` object is a player that wraps the BufferSourceNode from the Web Audio API that accepts the PatcherAudio as the input and can play the audio with loop and different playback speed.

The PatcherAudio API provides necessary information getters like its number of channels, length and sample rate; and manipulation methods like split, concatenate, pick, paste, remove, insert, etc.

Figure 4 shows an example of picking a section (from sample 24000 to 25000) of the existing buffer with 48000 samples.

[4] `https://www.ffmpeg.org/`
[5] `https://emscripten.org/`
[6] `https://webassembly.org/`
[7] `https://github.com/ffmpegwasm/ffmpeg.wasm`

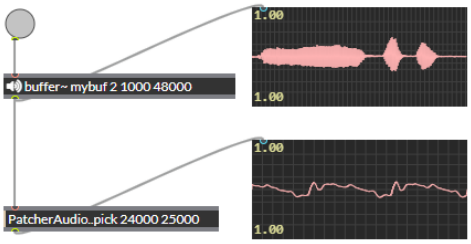

Figure 4. Audio buffer manipulation

4. AUDIO PLUGIN SUPPORT

4.1 WebAudioModule Plugin Standard

In the music production industry, most hardware devices have been substituted by software solutions over the past decades. Using DAWs with third-party audio plugins that act as synthesizers or audio effects is a common workflow for modern audio projects. VST (Virtual Studio Technology), introduced by Steinberg in 1996, is one of the most used cross-platform native plug-in formats based on C++. It provides a public SDK and API documents to allow plugin providers and host developers to implement them correctly.

Due to the limitation of the web platform, native audio plugins such as VSTs cannot be used in a web application without installing additional native software. Thus, for web-based DAWs, the need arose for a new standard of WebAudio plugins, offering similar functionality to their native counterparts.

In 2015, the first version of the Web Audio Modules (WAM) [10] standard is created, primarily for native plugins developers to port their existing plugins to the web. In 2018, they joined forces with other groups of people working on interoperable Web Audio plugins and plugin hosts to synchronize their efforts toward the beginnings of an open standard called Web Audio Plugins (WAP) [11, 12], covering a wider range of use cases. Recently, taking into account previous developments and the feedback received from developers over the past few years, a new version of the WAM standard has been created by emerging recent technological developments.

Now, a WAM plugin can be fetched from the web using a URI, and be initialized using the current WebAudio context. The plugin comes with a standardized API that can create a UI, get or adjust parameters, schedule events like parameter automation or MIDI messages, and connect with other WAMs or native WebAudio nodes.

4.2 WAMs in the Audio Editor

The Audio Editor in JSPatcher has a dedicated effect "rack" for WAMs. When the audio is loaded, the "rack" is empty so that users can add WAMs from their URIs. The audio will be played and processed through a pre-gain, then through the ordered WAM effects, finally through a post-gain. Users can use the rack UI to add or delete WAMs or display the WAM's own UI. (Figure 5)

Web Audio's OfflineAudioContext is an audio context

that, instead of process audio in real time, generates audio data from a WebAudio graph as fast as it can. When a user needs to export the entire audio file rendered with effects, or to apply the effects in-place, we will use a copy of the current "rack" in a new OfflineAudioContext to render the audio.

Figure 5. WAM plugins in the audio editor

4.3 WAMs in Patchers

It is also important to be able to create interactive audio programs with WAMs in JSPatcher. Using the `plugin~` object, users can load WAMs via a URI into a patcher.

When the `plugin~` receives a text string (as URI), it will remotely download the code from the URI and initialize it as a WAM. Then, it will automatically wrap the WAM and load its UI. Its inlets and outlets will be connectable with other WebAudio node objects. Additional inlets will be created to accept real time parameter change messages.

As WAMs support MIDI messages as input, the first inlet of the object accepts numbered arrays as MIDI event messages to the WAM inside. In addition, as WAMs have the ability to transmit non-audio events between each other, the patcher connection between two WAMs will be treated as a special one and make the necessary connection.

Figure 6 is an example of a WAM loaded in a patcher with an audio player.

Figure 6. WAM with an audio player in a patcher

5. COMPUTER-AIDED COMPOSITION

5.1 Context

Algorithmic composition often needs computer programs to calculate its musical representation. To design such a computer-aided composition (CAC) system for composers, visual programming and musical score display are two key features. OpenMusic, developed at IRCAM, shows the power of such a VPL in algorithmic composition with its composer-oriented design. The web environment, with its strong accessibility from device to device and flexibility from the UI perspective, is already a common platform for the low-code development system. It is likely to be highly suitable for a CAC system.

High-quality digital musical scores can be dynamically rendered and displayed on a computer using the SVG (Scalable Vector Graphics) format. Libraries like VexFlow, [8] Guido, [9] Verovio [10] or abcjs [11] can render music scores on the web. However, not all of them provide an API to interactively deal with the musical structure (model) behind the score.

The challenge we are facing is a need for a JavaScript-compatible musical model system that is able to:

1. Calculate and compose abstract music (like a MIDI file),
2. Play via WAM synthesizers,
3. Be displayed as a musical score.

Therefore, we created a JavaScript library called Sol for the calculation of different musical concepts. [12] It aims to deal with the musical model issue in the web-based CAC system. Then, a WebAssembly version of Guido [13] is used to render the score as it provides the AR (Abstract Representation) API to easily convert the musical model to the score. Finally, these feature are integrated as a package [14] into JSPatcher.

5.2 Musical Model

Musical notation is basically a representation of the musical structure based on a set of different musical concepts and a composition of instances of these concepts. To facilitate the CAC, the modeling of these concepts, or generally music theory, should be implemented in a digital way to allow the calculation between the musical concepts.

Yet, modern music theory is so complex and diverse that its modeling cannot be all-inclusive. The Sol library is only a proof of concept with a covering of very basic but necessary concepts in a common CAC workflow.

5.2.1 Pitch and Note

The difference between a pitch and a note can be ambiguous. In our library, "note" means different "pitch classes" in an octave. A note can be A, B, C, D, E, F and G with any number of sharp or flat accidentals.

[8] `https://www.vexflow.com/`
[9] `https://guido.grame.fr/`
[10] `https://www.verovio.org/`
[11] `https://www.abcjs.net/`
[12] Open-sourced on `https://github.com/fr0stbyter/sol`
[13] `https://github.com/grame-cncm/guidolib`
[14] `https://github.com/jspatcher/package-cac`

Text strings can be used to construct a Note object. For example, "C###" "Db" or "Bx" are recognizable as "note C with 3 sharps" "note D with 1 flat" and "note B with double sharps." Internally, the note name and the accidentals will be preserved and can be used for further calculation. Integer numbers can also be used to construct a note as the offset in semitones from note C, in this case, the note name and the accidentals will be determined automatically.

Pitch is an extended Note object with additional octave information. It is then a more concrete concept as the frequency can be calculated. It is also possible to get a pitch from a frequency with an approximation of a semitone. The pitch's offset follows the $C4 = 60$ standard.

Both Note and Pitch object supports some kinds of mathematical calculations. Adding or subtracting a number from a Pitch will change by semitones its offset. The offset difference can also be calculated between two Pitches For example, the following code creates a C4 pitch, by adding 12 to get a C5 pitch, then by subtracting a C4 pitch to get 12.

```
const pitch = new Pitch("C4");
pitch.add(12).toString(); // => "C5"
// Now pitch is C5
pitch.sub(new Pitch("C4")); // => 12
```

The multiplication between a pitch and a number is supported to calculate an approximated pitch based on the current pitch as the fundamental frequency and a multiplication ratio. The division is also possible between the pitch and a number or another pitch to calculate a new pitch or the frequency ratio between them. For example, the following code creates a C3 pitch, by multiplying 4 (to its frequency) to get a 2 octaves higher pitch C5, then by multiplying 1.5 to get a perfect fifth higher.

```
const pitch = new Pitch("C3");
pitch.mul(4).toString(); // => "C5"
// Now pitch is C5
pitch.mul(1.5).toString(); // => "G5"
```

5.2.2 Interval

Interval is the distance between two pitches, but it is more complex than just the number of semitones. With different note names and accidentals, the same distance, in equal temperament, can have different intervals such as diminished fourth and major third. Its calculation involves three properties: quality, degree and octave.

In Sol, an interval can be created with a text string, a frequency ratio, or from these properties. "A4" "d5-1" "P5" "M9" "m10+1" are usable strings for "augmented fourth" "diminished fifth with one octave lower" "perfect fifth" "major ninth (internally will be transformed to a major second with one octave higher)" "major tenth with one octave higher (internally a major third with two octaves higher)."

Here, the distance between two pitches can be "negative." In one octave, the interval between note F and note B is augmented fourth, and the interval between note B and note F will be "diminished fifth with one octave lower." The operation can be executed using the following code:

```
const b = new Note("B");
const f = new Note("F");
const i1 = f.getInterval(b);
i1.toString(); // => "A4"
const i2 = b.getInterval(f);
i2.toString(); // => "d5-1"
```

`i1` and `i2` from the code above are two intervals. They can be used for addition and subtraction from notes and pitches.

5.2.3 Chord

A chord is basically a set of pitches aligned vertically and executed at the same time. In our library, the Chord interface includes a base note or pitch and an array of intervals. It has methods such as inverse up and down, move up and down, etc. This model is chosen to easily move the chords, and also to solve the following question more easily:

"How to find the missing fundamental frequency of a given chord, if any?"

CAC is often used for spectral music composition algorithms, which involves the calculation of harmonics or the fundamental frequency. The *missing fundamental* of a sound [13] is an interesting topic in the scope. It has been demonstrated that the frequency of the pitch heard in response to a set of two or more successive harmonics corresponds to the greatest common divisor (GCD) of the harmonic set, even when there is no spectral energy at that frequency [14]. From a signal perspective, the autocorrelation algorithm can be used to find the periodicity of a sound and to determine the missing fundamental frequency. As we already have the frequencies from every pitch of the given chord, we can simply calculate the GCD of the frequencies.

However, the missing fundamental frequency should be an approximated value, as we are under the equal temperament, and human ears has a limited frequency discrimination capacity [15].

Therefore, we calculate a commonly approximated ratio between pitches from multiple ratios given by the intervals, we choose $\frac{1}{3}$ of a semitone, which is nearly a 2% difference, as a threshold of the frequency discrimination and the temperament compensation.

As a result, the algorithm is able to recognize that a dominant seventh chord has a ratio of $4:5:6:7$ and a missing fundamental on the two octaves lower dominant degree.

```
const chord = new Chord(
  new Pitch("C4"), new Pitch("E4"),
  new Pitch("G4"), new Pitch("Bb4")
);
chord.ratio; // => [4,5,6,7]
chord.phantomBase.toString(); // => "C2"
```

The code above calculates a dominant seventh chord on C4, it returns that its missing fundamental (`phantomBase`) is C2.

5.2.4 Duration

A musical value (duration) in fractions of beats can be expressed using the Duration object. Its constructor accepts a

fraction (numerator and denominator) or a text string such as “4n” “8nd” or “16nt” for “a quarter” “a dotted 8th” and “a triplet 16th.” Mathematical operations like compare, add, subtract, multiply and division are also available. For example, we create in the following code the length of a quarter note (one beat), then multiply it by 1.5 to get a dotted quarter, divide it by 9 to get a triplet 16th, finally add a quarter to it to get a length of $\frac{7}{24}$ beat.

```
const dur = new Duration(1, 4); // 1 beat
dur.mul(1.5); // 3/8 beat
dur.div(9); // 1/24 beat
dur.add(new Duration(1, 4)); // 7/24 beat
```

5.2.5 Sequence

Combining the presented models, a sequence can be formed with an array of chords or null values for a rest, and their duration. The sequence interface contains minimum data to render a score segment. It can also be compiled to a MIDI file with a time signature and BPM (beats per minute) information.

5.2.6 Others

Other musical concepts such as Velocity, Scale, Tonality, Track, Instrument, Genre, etc. are added to the library as well. However, these concepts are more complex and still need to be improved in future works.

5.3 Score Rendering

The Guido engine [16] is originally a C++ library aimed to compile plain text as GMN (Guido Music Notation) and to an SVG format file for the score. But it also has two intermediate steps, AR (Abstract Representation) and GR (Graphical Representation) that allow rendering a score from its API [17].

Through an Emscripten toolchain, the engine has been compiled to WebAssembly with most of its API and released as a JavaScript library at NPM (Node Packages Manager) online. To use the Sol library with the Guido API, we added a method `toGuidoAR` to the `Note`, `Pitch`, `Chord` and `Sequence` to perform function calls, construct the musical structure in the Guido AR.

In JSPatcher, the Guido engine runs in a separate thread using the Web Worker API. A `guido.view` object is available to accept a Guido AR instance and shows directly the compiled SVG score in the object UI.

5.4 Other CAC-related Objects

Manipulations of arrays (lists) of pitches, durations, velocities are common tasks in a CAC work. Like the numerous array-related functions in OpenMusic, we also need to provide them in JSPatcher.

Array versions of JavaScript operators are available as objects with the prefix “[]” like `[]+`, `[]*`, `[]&&`, which are array versions of “add” “multiply” and “logical and.” They accept an array as the first input. When the second input is an array, values in two arrays will be applied to the function respectively according to their index; otherwise, every value in the first array will be applied to the functions with the second value.

We also added some functions existing in OpenMusic like `x2dx` (calculate the difference between values in an array), `dx2x` (integrate an array), `permute`, `combinations`, `arithmSer` (create an arithmetic series), `fiboSer` (create a Fibonacci series), etc.

6. EXAMPLE

The following patcher (Figure 7) shows a CAC project combining the presented features.

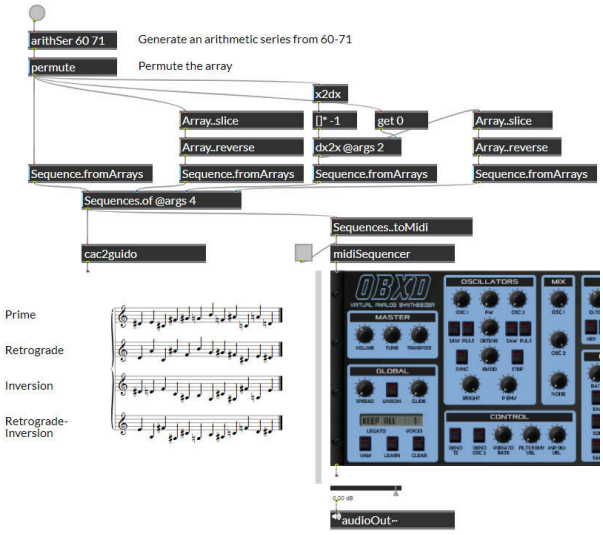

Figure 7. An example of CAC with a MIDI player

The first part of the patcher starts with a button that will create a Bang on click. The Bang generates an arithmetic series between 60 and 71, then randomly permutes the array to get the prime form of a dodecaphonic series. Its inversion is then calculated using the first value and the intervals between each value .The reversions of the two forms are calculated using a copy of the arrays. Using the Sequence and the Sequences functions, we now have a four-voice music segment. The score is displayed using the `cac2guido` object, and its MIDI file is generated and passed to a sequencer. Below the sequencer, a WAM synthesizer is loaded and waiting for real time MIDI messages. It produces audio data that is connected to the physical output. With the toggle, users can play or stop the sequence.

7. CONCLUSIONS

The scope of sound and music computing becomes wider today due to various possibilities to create music rapidly or even in real time. The development of machine learning and AI makes automatic composition possible and continuously inspires musicians to cooperate with the computer in the composition process. The accessibility offered by modern web technology with its highly active ecosystem is an opportunity for us to bring music programming and

CAC systems to a new stage. For instance, an artwork created on JSPatcher can be easily shared with people by designing a presentation mode of the patcher in which only selected boxes is showing with specific position and size, and specifying in the URI the location of the project file and the patcher filename, with additional options such as hiding the editor UI or make the patcher read-only. [15] In addition, UI components in patchers can also be interacted using touch devices which could make the design workflow simpler than existing VPLs on native platforms.

Today, audio processing in the web do have some limitations, especially the performance cost and the reliability compared to native C/C++ programs due to the implementation of different browsers, i.e. lack of optimization for multi-core CPUs. Even though, the web platform is still attractive to both developers and users. It has minimized the barrier for the deployment and use of any computer program from any device.

The described additions and improvements on JSPatcher are just the first steps towards a complete online IDE for sound and music computing. Feedbacks given by musicians from the community shows high interest in the future possibilities of JSPatcher such as supports for INScore, [16] collaborative editing and messaging via the network. As the project is open-sourced [17] with the examples and SDK (Software development kit) provided, contributions from third-party are feasible and welcome.

8. REFERENCES

[1] S. Ren, L. Pottier, and M. Buffa, "Build webaudio and javascript web applications using jspatcher: A web-based visual programming editor," in *Proceedings of the International Web Audio Conference*, ser. WAC '21, L. Joglar-Ongay, X. Serra, F. Font, P. Tovstogan, A. Stolfi, A. A. Correya, A. Ramires, D. Bogdanov, A. Faraldo, and X. Favory, Eds. Barcelona, Spain: UPF, July 2021.

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

[3] M. Puckette, "Pure Data," in *Proceedings of the International Computer Music Conference*. Thessaloniki, Hellas: The International Computer Music Association, Sep. 1997, pp. 224–227.

[4] J. Bresson, C. Agon, and G. Assayag, "Openmusic: Visual programming environment for music composition, analysis and research," in *Proceedings of the 19th ACM International Conference on Multimedia*, ser. MM '11. New York, NY, USA: Association for Computing Machinery, 2011, pp. 743–746.

[5] M. Puckette, "The patcher," in *Proceedings of the International Computer Music Conference*. San Francisco, United States: Computer Music Association, 1986, pp. 420–429.

[6] H. Choi, "Audioworklet: the future of web audio." in *Proceedings of the International Computer Music Conference*, Daegu, South Korea, Aug. 2018, pp. 110–116.

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

[8] J. Vilk and E. D. Berger, "Doppio: Breaking the Browser Language Barrier," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014, pp. 508–518.

[9] B. Powers, J. Vilk, and E. D. Berger, "Browsix: Bridging the Gap Between Unix and the Browser," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, 2017, pp. 253–266.

[10] J. Kleimola and O. Larkin, "Web audio modules," in *Proceedings of the Sound and Music Computing Conference*, Jul. 2015.

[11] M. Buffa, J. Lebrun, J. Kleimola, O. Larkin, and S. Letz, "Towards an open web audio plugin standard," in *Companion Proceedings of the The Web Conference 2018*, Lyon, France, Apr. 2018, pp. 759–766.

[12] M. Buffa, J. Lebrun, S. Ren, S. Letz, Y. Orlarey, R. Michon, and D. Fober, "Emerging w3c apis opened up commercial opportunities for computer music applications," in *The Web Conference 2020 DevTrack*, Taipei, Apr. 2020.

[13] A. Seebeck, "Beobachtungen über einige bedingungen der entstehung von tönen," *Annalen der Physik, 2nd ser.*, vol. 53, pp. 417–436, 1841.

[14] D. A. Schwartz and D. Purves, "Pitch is determined by naturally occurring periodic sounds," *Hearing Research*, vol. 194, no. 1, pp. 31–46, 2004.

[15] B. C. J. Moore, "Frequency difference limens for short-duration tones," *The Journal of the Acoustical Society of America*, vol. 54, no. 3, pp. 610–619, 1973.

[16] H. H. Hoos, K. Hamel, K. Renz, and J. Kilian, "The GUIDO notation format: A novel approach for adequately representing score-level music," in *Proceedings of the 1998 International Computer Music Conference, ICMC 1998, Ann Arbor, Michigan, USA, October 1-6, 1998*. Michigan Publishing, 1998.

[17] D. Fober, Y. Orlarey, and S. Letz, "Scores level composition based on the GUIDO music notation," in *Non-Cochlear Sound: Proceedings of the 38th International Computer Music Conference, ICMC 2012, Ljubljana, Slovenia, September 9-14, 2012*. Michigan Publishing, 2012.

[15] The URI for share could be like `https://fr0stbyter.github.io/jspatcher/dist/?projectZip=../../soundcraft/Soundcraft6.zip&file=020.jspat&runtime=1`

[16] `https://inscore.grame.fr/`

[17] `https://github.com/Fr0stbyteR/jspatcher`