Data-Driven Live Coding with DataToMusic API

Takahiko Tsuchiya & Jason Freeman
Georgia Institute of Technology
Center For Music Technology
840 McMillan St., Atlanta, GA 30318
takahiko@gatech.edu
jason.freeman@gatech.edu

Lee W. Lerner
Georgia Institute of Technology
Georgia Tech Research Institute
250 14th St. NW, Atlanta, GA 30318
lee.lerner@gatech.edu

ABSTRACT

Creating interactive audio applications for web browsers often involves challenges such as time synchronization between non-audio and audio events within thread constraints and format-dependent mapping of data to synthesis parameters. In this paper, we describe a unique approach for these issues with a data-driven symbolic music application programming interface (API) for rapid and interactive development. We introduce DataToMusic (DTM) API, a data-sonification tool set for web browsers that utilizes the Web Audio API[1] as the primary means of audio rendering. The paper demonstrates the possibility of processing and sequencing audio events at the audio-sample level by combining various features of the Web Audio API, without relying on the ScriptProcessorNode, which is currently under a redesign. We implemented an audio event system in the clock and synthesizer classes in the DTM API, in addition to a modular audio effect structure and a flexible data-to-parameter mapping interface. For complex real-time configuration and sequencing, we also present a model system for creating reusable functions with a data-agnostic interface and symbolic musical transformations. Using these tools, we aim to create a seamless connection between high-level (musical structure) and low-level (sample rate) processing in the context of real-time data sonification.

Keywords

Web Audio API, Data Sonification, Sample-Level Modulation, Real-Time Clock, Live Coding

1. INTRODUCTION

In recent years, web browsers have become a versatile platform for interactive multimedia applications. Many web pages, for example, integrate various data sources and real-time visual rendering to create interactive data visualizations[2]. Similar to visualization, data sonification, the use of non-speech audio to represent information [14], is a widely explored area for analytics, communication, and other purposes. Data sonification is used in both practical and artistic applications. In the latter, musicians have created algorithmic compositions and live performances with non-musical information, such as weather and sensor input [7, 8]. In recent years, more and more applications of data sonification have been deployed on-line for web browsers.

In a previous paper, we introduced the DataToMusic (DTM) application programming interface (API), a JavaScript tool set for data sonification for web browsers [13]. In the paper, we discussed the effectiveness of common musical structures and expressions for representing multi-dimensional data. Using the DTM API, we explored possibilities in data-agnostic models that can flexibly translate unknown data input to musical output. We created such algorithms by combining various analysis and transformation tools of the API as well as rendering methods including real-time notation with Guido [5] and audio synthesis and playback using the Web Audio API. We also developed a live coding [1] capability in the DTM API, allowing us to use it in a musical performance in addition to interactive development within a web browser. Live coding benefits performers and developers in data sonification in many ways. For example, it lets us experiment with the design of musical algorithms as we process a large data set or a data stream in real time. With immediate feedback on the design changes, it creates a "continuity between the old and the new behavior" [10] that enables fine tuning of designs without interrupting the musical flow in time and rhythm. Live coding also requires a robust modular framework in which we can safely connect and disconnect modules, facilitating the reconfiguration of a complex application easier.

Developing a real-time system capable of live coding, however, involves technical challenges. For instance, creating a precise and fail-safe clock in single-thread JavaScript is very difficult when we want to transform and map data to audio, automate synthesis parameters, and sequence audio events all in a synchronized manner. In addition, creating an interface for a simple data-to-parameter mapping can be challenging as we work with high-level sequencing as well as low-level audio processing that the Web Audio is capable of. In this paper,

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

[2]For example, http://www.nytimes.com/interactive/2015/us/year-in-interactive-storytelling.html

Web Audio Conference WAC-2016, April 4–6, 2016, Atlanta, USA

we discuss a runtime data-driven approach for audio synthesis and performance that takes advantage of the functionalities of Web Audio and addresses some of its limitations. This paper will describe the design problems and propose solutions for them in the context of implementations of the modules of our API. In the following section, we will review the literature and compare prior approaches to our approach. Section three introduces the DataToMusic API, focusing on both the implementations of synthesizer and clock modules and the adaptive musical model, which may be applied to symbolic as well as timbre-level transformations.

2. RELATED WORK

In the last few decades, interactive and real-time coding in native environments has become increasingly popular among multimedia artists and developers with popular tool sets such as Max/MSP[3], SuperCollider[4], Chuck[5], and many more. Compared to native tool sets, web tool sets often offer less comprehensive but more specialized or characteristic functionalities, and bring higher accessibility for general users and developers. For example, Gibber is an audio-visual live-coding environment [9] with a high flexibility for parameter mapping and automation. It takes an "everything is a sequence" stance, with which enables easy sequencing of any property or method of any Gibber object[6]. Using Gibber's audio engine Gibberish API[7], BRAID allows us to interactively construct musical instruments with graphical interfaces and to quick configure the synthesizer using an in-line code editor [12]. Another audio live-coding application is Wavepot[8], which automatically evaluates the changes of code at a musical interval to provide real-time and incremental feedback for design changes. Lissajous[9] allows multi-track musical sequencing with rapid chainable methods, utilizing the browser JavaScript console for read-eval-print-loop (REPL) based live coding. Another web API capable of live coding is EarSketch, an on-line programming education system based on music remixing [6]. In EarSketch, while the audio graph is constructed at user-script compilation, it supports live coding in the form of quick re-compilation of audio tracks during a playback. This live-coding approach is effective for the real-time manipulation of audio events with a minimum downtime between the events.

These different applications use or combine runtime development paradigms such as just-in-time (JIT) compilation [10], REPL, or selective line evaluation, functional programming that allows the dynamic creation and handling of functionalities and algorithms as well as the automatic update of the audio graph. Collins suggests the dynamic restructuring of the audio graph as the main principle of audio live coding, as found in Max/MSP and SuperCollider [1]. The DTM API extends this idea with real-time data processing and mapping for creating complex audio expressions using JIT compilation as well as intervalic evaluation of code.

[3]https://cycling74.com/
[4]http://supercollider.github.io/
[5]http://chuck.cs.princeton.edu/
[6]https://www.gitbook.com/book/bigbadotis/gibber-user-manual/details
[7]http://www.charlie-roberts.com/gibberish/
[8]http://wavepot.com/
[9]http://lissajousjs.com/

3. DATATOMUSIC API

3.1 Overview

DataToMusic API is a JavaScript library for data-agnostic sonification in web browsers. We originally developed this tool set to experiment with symbolic musical structures and create reusable models for varying data formats. When creating an application of sonification, our dataset or the data stream usually has a unique dimensionality, cardinality, types, and value ranges. Integrating a specific data format can lead to the design of an audio- and data-mapping scheme that is not easily reusable; and its audio or musical expressivity may also depend on particular data.

To address such problems, researchers of sonification have proposed reusable design frameworks such as model-based sonification (MBS) [4] and parameter mapping sonification (PMSon) [3]. MBS offers a high interactivity and generalizability with acoustic modeling. Nonetheless, it can be computationally demanding for web-browser-based implementations, and it is also mainly specialized in timbral rather than musical expressivity. PMSon recommends strategies for generalized data preprocessing, analysis, and mapping procedures for data-to-sound synthesis. While this technique is widely accepted, the mapping of a PMSon system may not be compatible with various data sources.

Although DTM was inspired by PMSon in the audio synthesis domain, it focuses on creating a model structure that adaptively maps data input to parameters of a musical structure, providing uniform mapping interface similar to the "flowboxes" of UrSound proposed by Essl [2]. The following code example shows the default adaptive mapping models of DTM, which take a single-dimensional array of any type, convert the type (e.g., for a character array, it may be encoded into a numerical representation such as bag-of-words vector, ordered by the frequency), and normalize, while preserving the domain range of the input and re-sampling into a target length if specified (Code Example 1).

```
// Default mapping models that convert an
   input array (of any type) into a
   normalized numeric array.
uni = dtm.model('unipolar'); // 0 to 1
bi = dtm.model('bipolar'); // −1 to 1

// Create a synthesizer object
s = dtm.synth().freq(440).play()

// Synth amp modulated with 'hello'
s.amp(uni('hello').fit(16, 'linear'))

// Pan modulated with a linear envelope
s.pan(bi([1,3,2,5]).fit(1000))

// Random values with the length of 1024.
random = dtm.gen('random').size(1024);
s.wavetable(bi(random));
```

Code Example 1: Adaptive Mapping Models

Table 1: The Main Modules in DTM

Data Structure	Real-time Operations	Model Abstracts	Output
dtm.data, dtm.array dtm.gen	dtm.master, dtm.clock	dtm.model, dtm.instr	dtm.synth, dtm.osc, dtm.guido

For creating an adaptive mapping interface, transformation functions, and other real-time processing features, the DTM API includes various modules categorized as follows: data structures, helper functions, real-time event handlers, model abstracts, and output and renderers (Table 1). In this paper, we mainly focus on the dtm.synth (output) and dtm.clock (event handler) modules that together integrate the Web Audio API in a novel approach.

3.2 Synthesizer Implementation

In designing and developing the synthesizer class, we examined a few unconventional approaches to achieve a balance among the ease of data mapping, the modularity of audio effects, and the sample-level operation on the audio event with high-resolution automation and sequencing. The dtm.synth module is essentially an interface to the Web Audio API that offers real-time audio synthesis and a flexible audio graph environment. Writing out instantiations and connections of nodes directly in Web Audio, however, can become quite verbose and is not suitable for rapid development or live coding scenarios. The dtm.synth instead provides simple chainable methods for constructing as well as reordering audio nodes (which may or may not consist of the default Web Audio nodes such as a DelayNode) by simply moving the insertion point of a method call (see Code Example 2). Similar techniques of dynamic construction of an audio effect chain are found in other Web Audio applications such as EarSketch.

```
// Create a note
var s = dtm.synth().play()

// Set the wavetable (a square wave)
s.wavetable([-1,1])
// Pre-rendering delay effect
s.delay(0.3)
// Another delay for comb-filtering
s.delay(0.9, 0.001, 0.8)
// A low pass filter.
s.lpf(2000)
// Post-rendering sample-level effect
s.bitquantize(8)
// Post-rendering LPF effect
s.lpf.post(5000, 1)
// Panning only applied at the post-
   rendering stage
s.pan(-0.2)
```

Code Example 2: Chaining Audio Effects

In the dtm.synth, we can apply custom effects to an audio event at the sample level without the ScriptProcessorNode while operating with data in “real time”. This operation is done by utilizing the automation methods such as the setValueCurveAtTime and multiple offline audio contexts of the Web Audio. The basic framework of audio synthesis and parameter mapping in the dtm.synth consists of two distinct phases: an off-line rendering of audio events followed by a real-time playback and processing of the rendered clip (see Figure 1). In the first phase, the basic parameters such as the wavetable, amplitude, and frequency are created with default values in an instance of the OfflineAudioContext. The off-line events, including the pre-rendering effect chain, are processed with parameter automations, and then passed to a new BufferSourceNode for real-time playback. The pre-rendering effects can be such as a delay, a filter, or a ring modulator and a frequency modulator that utilize audio-rate (or control-rate depending on the parameter of an AudioNode) modulation of the setValueCurveAtTime function.

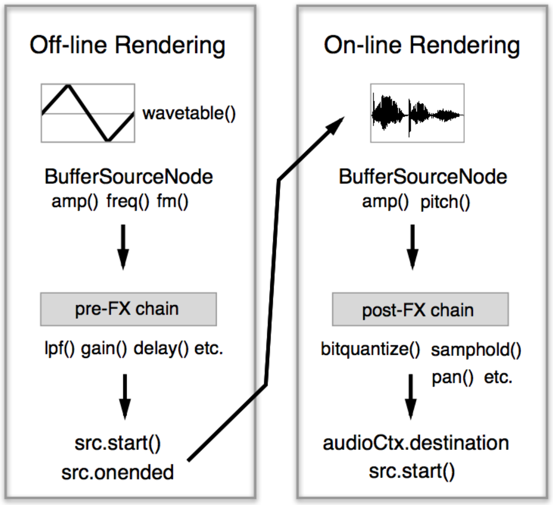

Figure 1: Audio Event Overview

In the second stage, post-rendering audio effects are applied to the rendered buffer. The post-rendering effects may include the similar types from the pre-rendering effects, but also adds sample-level operations such as a bit quantizer and wave shapers that directly modify the rendered buffer. This structure, therefore, allows us to apply custom audio effects either in real time or an instantaneous manner. The two-fold rendering is especially effective with wavetable synthesis, in which one may want to apply sample-level effects to the wavetable itself (that is typically very short) as well as to the resulting audio from parameter automations with a longer duration.

Another design challenge for the dtm.synth was interfacing the data input to the Web Audio synthesis and parameter curves. In our previous implementation of the dtm.synth, the audio synthesis parameters, such as an oscillator’s frequency (a “number” type), a wavetable (a PeriodicWave generator using a numerical array), and an amplitude envelope ([A, D, S, R]), all accepted non-uniform data structures. Many of the parameters had a single-value interface, and they were automated with the setTargetAtTime method triggered by the real-time clock. With this interface, one could take some values from a data source to modulate various parameters, but it lacked in flexibility of mapping any length of sequence to a parameter, or synthesizing and modulating at a higher rate and with complex curves. In addition, the timing of real-time clock for updating parameters was also not reliable enough for precise sequencing. To address these issues, we took a completely new approach for parameter mapping and automation. In the new version, the dtm.synth uses a variable-length Float32Array for every modulatable parameter, including the wavetable for the oscillator. Compared to the previous single-value mapping interface (which still can be done by using a single-value array), this allows more direct mapping of data points to time-ordered events, close to linear value-to-value mapping. One can process complex curves with a large number of data points (e.g., 10,000 or more) using the dtm.array or generate simple shapes as a LFO with a few data points and map to any parameter.

```
// Create a synth object
var s = dtm.synth().play();

// Create a wavetable with array generator
var someSteps = dtm.gen('noise').size(10);

// Stretch the wavetable into the length
    of 3000, using the cubic interpolation
s.wavetable(someSteps.fit(3000, 'cubic'));

// Generate a pattern, rescale and
    quantize
var melody = dtm.gen('fibonacci')
  .size(10).range(60, 90).round();

// Assign to the MIDI pitch with some
    transformation
s.notenum(melody.repeat(2).mirror());

// Set the base amplitude
s.amp(0.5);

// Modulate the base amplitude with
    repeating ramps
s.amp.mult(dtm.gen('decay')
  .repeat(melody.len))
```

Code Example 3: Mapping Arrays to the dtm.synth

For automating parameters in real-time, we tested and implemented two approaches: one with a single call of the setValueCurveAtTime, and the other with the setValueAtTime called for every data point. The setValueCurveAtTime method is beneficial in several ways; for example, it automatically fits an array to the target length, and it can modulate a parameter at a higher rate, as described above. It has, however, limitations in the current browser implementations in terms of the interpolation method and synchronization of array data points to time. For the value interpolation, as opposed to a linear interpolation specified in the API documentation, it only applies a step interpolation to the array. The resulting stepped value curve also is shifted in time when applied, causing an unwanted rhythmic offset.[10] The setValueAtTime method, in contrast, works more reliably for time synchronization. Therefore, by default, the dtm.synth first tries to use the setValueAtTime. As both automation methods do not provide a linear interpolation for all browsers, it is expected for the user of the DTM API to utilize the fit and stretch functions of the dtm.array when mapping it to a parameter of the dtm.synth. These functions re-sample the input array into the target length with interpolation methods such as linear, step, cubic, fill-zeros, and many other. With a relatively lower number of data points (up to a few thousand per second), the setValueAtTime method works reliably and precisely. However, when the number of data points is larger or even exceeding the duration of the audio event in samples, a large number of the setValueAtTime being scheduled in the process starts to cause delays in the main and the audio threads. In such cases, the dtm.synth automatically switches at a certain threshold to use the setValueCurveAtTime method for less computation but with less timing accuracy.

[10]In addition to these limitations, the audio-rate modulation of the setValueCurveAtTime does not work well in FireFox. These issues were present in Chrome, FireFox, and Safari in the late 2015. As of February, 2016, Chrome has the linear interpolation behavior implemented.

3.3 Clock Implementation

When programming real-time musical algorithms and applications, a clock generator is likely to be essential for playing musical notes and processing other events in a synchronized manner. The DTM API, in fact, heavily relies on clocks for audio synthesis, creating rhythmic structures, processing data such as streaming and block-wise querying, and live-coding operations. Despite the many attempts to implement a precise and robust clock in browser JavaScript, implementation has always been difficult with the limitation of single-thread operation, which may randomly delay a clock callback because of other heavy computations such as rendering of visual elements. We try to implement a clock system that minimizes such artifacts on the audio synthesis and the rhythmic performance of audio events by utilizing the Web Audio schedulers and error compensation with a lookahead time for the dtm.synth.

In our earlier implementation of the DTM API, we experimented with the behaviors of callback clocks with the setInterval, the ScriptProcessorNode, the onended EventHandler with audio source nodes, and the requestAnimationFrame method. As discussed by Wilson [15], the delay caused by the main-thread operations is highly unpredictable, directly affecting the callback timing. Even using the audio-thread timer and callback with the Web Audio functions such as the BufferSourceNode, OscillatorNode, and ScriptProcessorNode, the main-thread delays cannot be isolated, and it adds a complication of correcting the timing gap caused by buffer-based audio processing. For implementing a re-schedulable clock with the Web Audio, the most common approach may be to combine a lower-rate main-thread clock for short-segmented scheduling and a higher-rate and higher-precision audio-thread scheduler for Web Audio events [15, 11], with optional overlaps for overcoming unpredictable delays on the main thread. This technique is effective in managing tempo changes in real time, but its application is basically limited to Web Audio events as we cannot precisely synchronize main-thread functions with the Web Audio events. In order to synchronize the audio events, rhythmic structure generated from data, and constant array processing, we employed a similar approach to the above-mentioned twofold clocks with additional functions such as process deferring and lookahead delay for a timing error compensation, master-slave synchronization, and the callback management for live coding. For clock synchronization, the real-time master clock runs at the highest resolution for a given tempo, and the tick of a slave clock is triggered at a specified lower-rate. This allows multiple instances of dtm.clock with different rates to be synchronized. In the musical context, a synchronized clock is typically used at a fixed rate between a quarter note to a few measures with the dtm.synth audio events. In this rather large interval, a precisely-timed sequence of Web Audio events can be generated using a single note event or a set of notes with specified delays. This callback clock is also used in other output formats such as real-time musical notation and note list, as shown in Section 3.1.

As mentioned above, the dtm.synth utilizes the lookahead value of the dtm.clock for error compensation. The API connects these objects in an automatic and context-aware manner. That is, once a dtm.synth object is instantiated within a dtm.clock's callback function, the synth object locates the parent clock in the context[11] and retrieves its look-ahead value as well as the tick interval. From these, the dtm.synth calculates the starting time of the audio processes and the event duration for parameter modulations (see Code Example 4). In particular, the lookahead period is used to defer the first off-line buffer rendering until all the array operations (and other heavy computations) are resolved, then the second on-line rendering is played at a delayed timing using the specified lookahead value. Besides this automatic time adjustment, it is also possible to separate the clock and synth and assign an external clock used in another synth object to synchronize the audio events together.

```
// Generate a decaying envelope with the
    length of 1000.
var env = dtm.gen('decay').size(1000);

dtm.clock(function () {
  // Note duration set to 0.25 seconds.
  dtm.synth().play()
    .notenum(60).amp(env);

  // Duration may also be manually
      specified.
  dtm.synth().play().dur(2.0)
    .notenum(67).amp(env);

  // Set the clock behavior
}).lookahead(0.1).bpm(120).time(1/8);
```

Code Example 4: Automatic Duration and Lookahead

In the context of live coding, the clock may be used for periodically (re-)evaluating the entire or a selected part of a script as well as managing the registered callback functions. It keeps track of named and anonymous callback functions using either the function name or the whole (stringified) structure of the object, detects live modification in them, and selectively retains, updates, or clears them. This helps prevent registering the old and new versions of a callback function separately.

[11] This is done by using the Function.caller.arguments property.

3.4 Live Coding and Mapping Complex Sequences

The dtm.synth and dtm.clock modules, therefore, allow a complex sequence of audio events constructed in (almost) real time with a sample-rate parameter modulation by data. A parameter curve may contain values from a single data point to thousands of data points that can be time scaled dynamically with the dtm.array transformation functions (i.e., using up or down sampling with various interpolation methods), which then is fit into the total sample length of the audio event (Code Example 5).

```
// Load an offline data set.
dtm.data('sample.csv', function (d) {
  // Get a column by the index.
  var data = d.col(0);

  // Create an exponentially decaying
      curve from 1 to 0
  var env = dtm.array([1,0])
    .fit(1000, 'linear')
    .expcurve(100);

  // Random jitter between 0 to 0.3 of the
      length 100
  var sus = dtm.gen('random', 0.3)
    .size(8).fit(100, 'cubic');

  env.concat(sus);

  var s = dtm.synth().play().amp(env)

  // clone() allows multiple edits from
      the same source
  s.wavetable(data.clone().range(-1,1))
  s.freq(data.clone().range(1000, 8000)
    .logcurve(200).fit(16))
  // Downsample into the length of 16 (a
      typical musical beat length).
  s.bitquantize(data.clone().range(16,2))

  var sin = dtm.gen('sine').size(32);
  s.lpf(sin.range(200,2000).logcurve(30));
});
```

Code Example 5: Mapping Data to a dtm.synth

One concern, however, is that the time scale of the parameter curve is always relative to the duration of an audio event (set by the clock interval or the duration parameter of the dtm.synth), which may require the user's attention on the resulting speed of modulation for the temporal or rhythmic alignment in a musical structure. Another potential inconvenience is that the dtm.synth expects a certain range of numerical values for each parameter. A data input needs to be, therefore, converted accordingly to the input data type, range, distribution, as well as the synth parameter ranges. Such requirements of appropriating data format for various parameters is sometimes not ideal for live-coding situations, as it slows down the design process, data (re)mapping, and may also cause semantic errors. We can automate the mapping process by using the previously-mentioned model system as a simple scaler and type converter (Code Example 6).

```
// Create a model object
var freqModel = dtm.model('array')
  // Specify the type conversion method
  .toNumeric('histogram')
  // Modify the preset behavior
  .domain(function (a) {
    freqModel.params.domain = a.get('
        extent');
  })
  // Default behavior: freqModel(data)
  .output(function (a, c) {
    return a.range(20, 200)
      .logcurve(30)
      .fit(c.get('div'));
  });

// Create a self-repeating note
dtm.synth().play().repeat()
  .freq(freqModel(data.block(100)));
```

Code Example 6: Creating a Model for Synth Parameter

Lastly, although a single audio event with the dtm.synth may be able to create a rich musical expression using the data-driven parameter automation, one can create furthermore dynamic expressions with rhythmic, melodic, or harmonic sequences with audio events. Code Example 7 shows a simple sequencing model for making the specific beats to be delayed for creating a swing effect.

```
var swing = dtm.model()
  // Modify the default method call
      behavior
  .output(function (clock) {
    if (clock.get('beat') % 2 === 0) {
      // No delay for the down beats
      return 0;
    } else {
      // Delay the up beats by 15%
      return clock.get('interval') * 0.15;
    }
  });

dtm.clock(function (c) {
  var delay = swing(c);
  // Delay the playback timing
  dtm.synth().play().offset(delay);
});
```

Code Example 7: Creating a Swing-Rhythm Model

Another approach for rhythmic sequencing is to modulate the interval and duration of self-repeating synth notes.

```
// Create a self-repeating note
dtm.synth().play().rep(Infinity)
  .amp(dtm.gen('decay').expcurve(10))

  // Each note's duration is randomized
  .dur(dtm.gen('random',0,0.5).size(8))

  // The onset intervals alternates
      between these values
  .interval([0.5, 0.3])

  // Each note can contain a complex pitch
       modulation
  .notenum(data.fit(16))
```

Code Example 8: Rhythmic Sequencing Without Clock

4. CONCLUSION

We presented our approaches for implementing a data-driven interface for a Web-Audio-based synthesizer, as well as a real-time clock system for controlling audio and non-audio events with fewer timing errors. In addition, using the sample-level mapping and musical structure models, we described possibilities of complex musical expressions in both symbolic and timbre-level time scales. We experimented with these features in the DataToMusic API, a data sonification library for web browsers capable of live coding. The DataToMusic API is publicly available as a GitHub repository[12], and as a demo application for on-line live coding[13].

5. REFERENCES

[1] N. Collins. Generative music and laptop performance. *Contemporary Music Review*, 22(4):67–79, 2003.

[2] G. Essl. *Ursound: Live Patching Of Audio And Multimedia Using A Multi-Rate Normed Single-Stream Data-Flow Engine*. Ann Arbor, MI: MPublishing, University of Michigan Library, 2010.

[3] F. Grond and J. Berger. Parameter mapping sonification. *The sonification handbook*, pages 363–397, 2011.

[4] T. Hermann. Model-based sonification. *The Sonification Handbook*, pages 399–427, 2011.

[5] H. H. Hoos, K. A. Hamel, K. Renz, and J. Kilian. *The GUIDO Notation Format: A Novel Approach for Adequately Representing Score-Level Music*. 1998.

[6] A. Mahadevan, J. Freeman, B. Magerko, and J. C. Martinez. EarSketch: Teaching computational music remixing in an online Web Audio based learning environment. 2015.

[7] A. Polli. Atmospherics/Weatherworks: A Multi-Channel Storm Sonification Project. In *ICAD*, 2004.

[8] T. Riley and K. Quartet. Sun Rings, 2002.

[9] C. Roberts and J. Kuchera-Morin. *Gibber: Live coding audio in the browser*. Ann Arbor, MI: MPublishing, University of Michigan Library, 2012.

[10] J. Rohrhuber, A. de Campo, and R. Wieser. Algorithms today - Notes on language design for just in time programming. *context*, 1:291, 2005.

[11] N. Schnell, V. Saiz, K. Barkati, and S. Goldszmidt. Of Time Engines and Masters. 2015.

[12] B. Taylor and J. Allison. BRAID: A Web Audio Instrument Builder with Embedded Code Blocks. 2015.

[13] T. Tsuchiya, J. Freeman, and L. W. Lerner. Data-to-Music API: Real-Time Data-Agnostic Sonification with Musical Structure Models. *Proc. of the 21st Int. Conf. on Auditory Display*, 2015.

[14] B. N. Walker and M. A. Nees. Theory of Sonification. In *The Sonification Handbook*. Berlin, 2011.

[15] C. Wilson. A tale of two clocks: Scheduling Web audio with precision. 2013.

[12]https://github.com/GTCMT/DataToMusicAPI
[13]http://dtmdemo.herokuapp.com/