Siren: Interface for Pattern Languages

ABSTRACT

This paper introduces *Siren*, a hybrid system for algorithmic composition and live-coding performances. Its hierarchical structure allows small modifications to propagate and aggregate on lower levels for dramatic changes in the musical output. It uses functional programming language *TidalCycles* [13, 14] as the core pattern creation environment due to its inherent ability to sustain uninterrupted audio output even with an incorrect syntax. Borrowing the best from Tidal, Siren augments the pattern creation process by introducing various interface level features: a multi-channel sequencer, local and global parameters, mathematical expressions, and pattern history. It presents new opportunities for recording, refining, and reusing the playback information with the pattern roll component. Subsequently, the paper concludes with a preliminary evaluation of Siren in the context of user interface design principles [18], which originates from the cognitive dimensions framework for musical notation design [6, 20].

Author Keywords

Siren, hierarchical structure, algorithmic composition, pattern programming, instrument design, live-coding, pattern roll

CCS Concepts

•**Applied computing** → **Sound and music computing;** Media arts; •**Human-centered computing** → *Information visualization;*

1. INTRODUCTION

For musicians who compose and/or perform with a personal computer as their main instrument, a number of different user interface paradigms are available. The digital audio workstation (DAW) with a graphical user interface (GUI) that is designed to be accessible and intuitive is arguably the most popular among these paradigms. However, with DAWs designed for accessibility, workflows for composing music rely extensively on navigating graphical user interface elements, and live performances are often predicated on extensive preparation. As such, computer musicians who desire to emphasize aspects of *liveness* [22], *flow* [17], and *virtuosity* in both composition and performance often turn to trackers and live-coding [7, 20, 19]. While trackers afford virtuosity, most are geared towards composition rather than being effective tools for improvisation and performance out-of-the-box, due to limited integration of synthesis capabilities. On the other hand, live-coding can be a powerful paradigm for performances, but setting up the toolchains required for live-coding and learning the intricacies of programming languages can be inconvenient for many musicians.

We introduce Siren, a performance and composition environment that marries a tracker-inspired user interface design with a hierarchical structuring of textual code blocks for defining musical patterns (Fig. 1). As such, it represents a hybrid approach to musical composition based on a fusion of the tracker paradigm and a purely textual, programming-oriented notation. This approach lowers the barriers to entry for live-coding while enabling for a more intuitive visual representation of compositions.

The design of the system is informed by previous works in the fields of programming and music. In this paper, we first review these strands of research that provide the background for our work and situate it in the context of research on interfaces for musical expression. Thereafter we describe the details of Siren's design and implementation, and conclude with a discussion on how we have utilized Siren as a powerful engine for live performance and composition.

Siren is available online as a free and open-source software[1] published under the GNU General Public License v3.0[2], and accepts contributions .

2. BACKGROUND RESEARCH

The underlying concepts and user interface design in Siren have been influenced by previous works. This includes research in the psychology of interaction with notations from the perspective of programming and music, and previously developed musical creation software and practices surrounding their use.

2.1 Notation and User Interface Design

One approach in previous research that guides the design of information artifacts—in particular, software for interacting with musical and algorithmic notation—is predicated on breaking down different aspects of the user experience into various cognitive dimensions. Based on notions introduced by Green and Petre [9] for investigating the experience of interacting with programming languages, researchers have developed a framework of cognitive dimensions to evaluate and guide the design of music notation systems and musical user interfaces [5, 6, 18, 20]. This framework also serves as the basis for design heuristics for supporting virtuosity

[1] `https://github.com/<anonymousForSubmission>/Siren`
[2] `https://www.gnu.org/licenses/gpl-3.0.en.html`

NIME'18, June 3-6, 2018, Blacksburg, Virginia, USA.

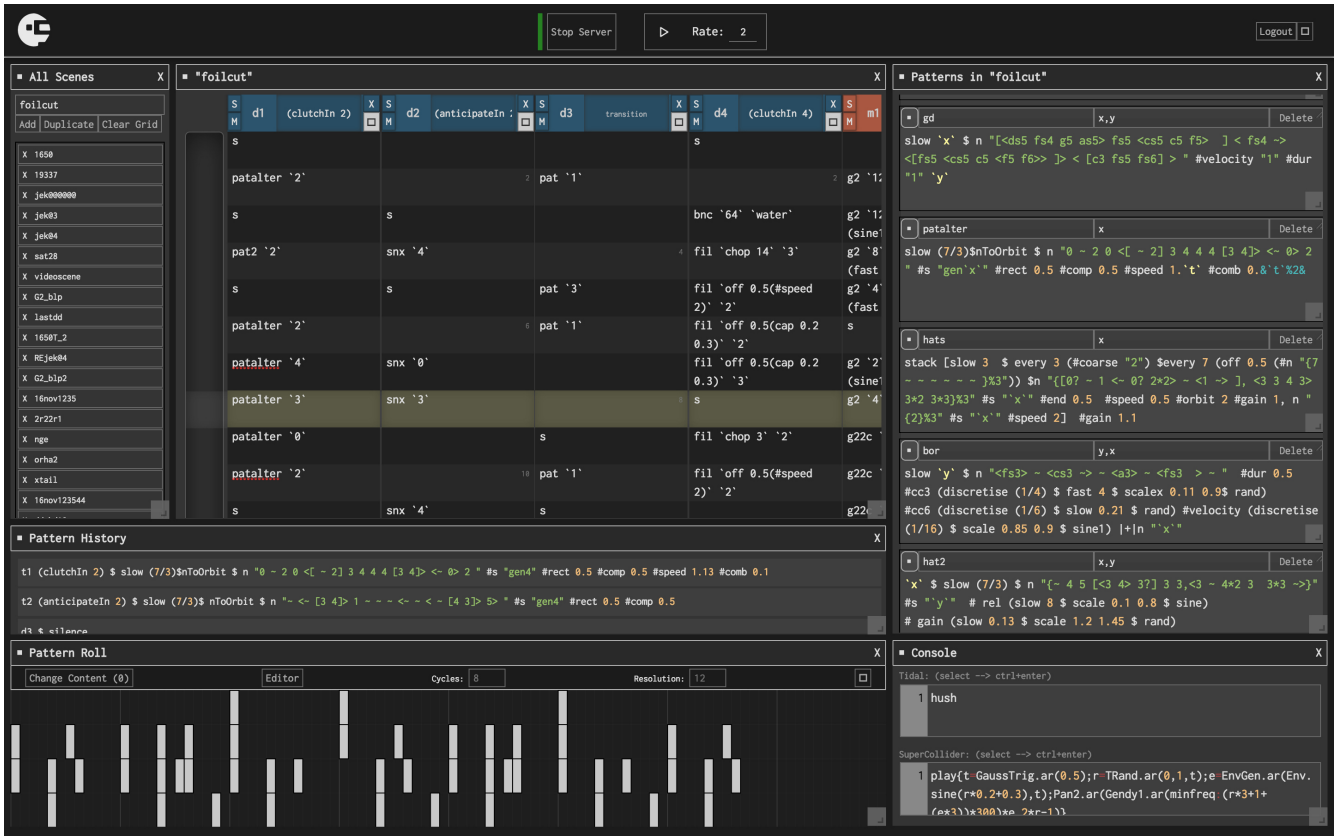

Figure 1: The user-interface – (top-left) the list of scenes; (middle) the multi-channel sequencer; (top-right) pattern dictionary; (middle-left) pattern history per channel; (bottom-left) pattern roll; (bottom-right) the text-based compiler

and flow states in musical user interfaces, and distilled into a set of questions that apply to such designs [21]. These approaches have been used to comparatively investigate a wide variety of notations and and interactive systems for musical creation, including traditional paper scores, graphical audio synthesis programming (Max or PureData), and DAWs. The scope and space in this paper does not allow for a comprehensive discussion of this body of research and how it is manifest in specific features in Siren, but a few of the most prominent aspects are unpacked in Section 4.

2.2 Trackers

The tracker user interface depicts notation as rows of discrete musical events positioned in columnar channels. Each cell in a channel can hold a note, parameter change, an effect toggle and other commands. Different patterns or loops can have independent timelines, which can be organized into a sequential master-list to form a complete composition. The tracker paradigm has been found in the aforementioned literature to be a highly effective digital instrument that supports virtuosity, liveness, and flow states while composing.

The earliest implementations of the tracker concept were released for the AmigaOS platform in the late 80s and early 90s, in applications such as Ultimate Soundtracker (1987) [8], NoiseTracker (1989) and Protracker[3] (1990). (A comprehensive history of tracker software can be found online, courtesy of the Tracker History Graphing Project[4].

2.3 Live-coding

[3]`https://sourceforge.net/projects/protracker`
[4]`http://helllabs.org/tracker-history/`

An early example of networked computer music platforms is SuperCollider (1996), an environment for audio synthesis and algorithmic composition that relies on the Open Sound Control (OSC) protocol. More recently, Conductive [2], offered a higher-level abstraction that uses density look-ups for the pattern programming [3]. As an interactive programming environment for live-coding, Foxdot [12] provides a fast and user-friendly abstraction to SuperCollider. Finally, TidalCycles [13, 14] introduced an embedded Domain Specific Language (eDSL) for composing patterns as higher order structures with a highly economical syntax. Most of these languages allow using the interpreter for the Glasgow Haskell compiler (GHCi) to trigger a synth by communicating with SuperCollider via the OSC protocol [15].

Siren utilizes Tidal as its main pattern programming structure, and constructs a strategically designed user-interface around it. Main concepts regarding the software and interface design emerge from pragmatic reasons such as keeping past playback data, parameterization, and abstractions. We created a back-end structure for patterns to achieve a fusion between the affordances of TidalCycles and SuperCollider, while allowing both to be used as the main pattern language in Siren.

3. PATTERN CREATION AND SEQUENCING WITH SIREN

Siren is based on a hierarchical structure of data, and a tracker-inspired user interface, initially intending to build on the concepts and technology of Tidal. The main idea is to support a hybrid interaction paradigm where the musical

building blocks of patterns are encoded in a textual programming language, while the arranging and dispatching of patterns is done via a grid-based user interface inspired by musical trackers [19]. Here, the concept of a *pattern* is borrowed from the Tidal's *pattern language*. It offers means to represent encoding of musical patterns, a library of pattern generators and combinators, a scheduling system for dispatching events. By augmenting the patterns, Siren uses *pattern functions* that are stored in a *pattern dictionary*. These pattern functions can be called from the cells of the tracker grid along with their optional parameters.

The creation of a pattern introduces a rhythmic element or a cycle. Sequencing the code allows the user to break out of linearity by modifying the cycle with different variables and transitions while allowing events remain coherent in their single and linear timeline. We implemented a timer based on a concurrent tick mechanism, generated from Ableton *Link* [1]. This allows events to be compiled synchronously and provides a precise timing mechanism in-between the patterns and other time-related modules (i.e. pattern roll, and global modifiers).

In the tracker grid, per convention, each column represents a *channel*. Channels here are analogous to tracks in contemporary DAWs. Each cell in a channel can be filled with a single *function call*, and its corresponding pattern function is located in the pattern dictionary. As such, cells in channels contain only the call for a function, which, as is conventional in many programming languages, comprises its name and the parameters to be passed with it.

A function that is defined in the pattern dictionary can be called from any cell of any channel. The execution happens as follows: The timer scans each channel from top to bottom, triggering pattern functions in cells as they are encountered. On trigger, Siren performs a look-up in the pattern dictionary. Once the desired pattern is found, the pattern is called by parsing its parameters and replacing them with the user input provided in the calling cell. If the related pattern in the dictionary is reconstructed correctly, the channel's interpreter or compiler executes the function (see Fig. 3).

3.1 Hierarchical Composition

The main musical structure in Siren is a *scene*, which acts as a top-level container and a framework for a composition [10]. Each scene can be thought of as a grid where each of its columns are the channels and rows are temporal tracker steps. The timer cycles through the rows, from top to bottom, and triggers the content of each cell. Each scene has a pattern dictionary for the storage of pattern functions, their parameters and implementations. Instances of these pattern functions can be written into the grid to act as a function call to patterns on trigger. A user can create multiple scenes with unique sets of pattern functions and channels.

3.2 Modules

The layout of Siren is designed to operate in a modular fashion with modules that contain specific elements of the system (see Fig. 1). The user can choose to toggle visibility of these modules at will. It is also possible to store four customs layouts with different modules. For example, one layout can maximize the channel module to take up the full screen, allowing the user to focus on sequencing, or the *console* could be made to take up the full screen for a user experience similar to a text editor, favored by programmers. Navigating between the layouts is also possible using convenient key-bindings.

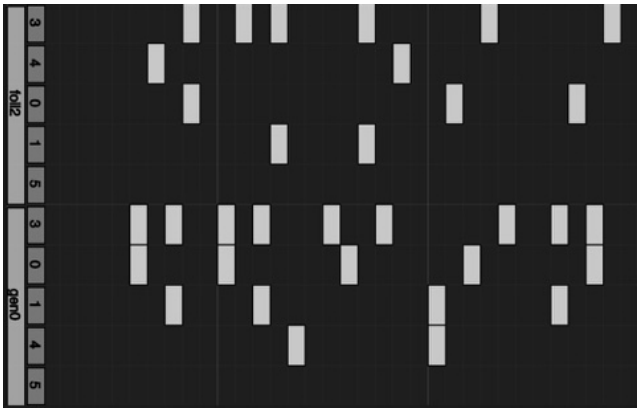

Figure 2: Close-up view of the pattern roll module – Horizontal dimension denotes time (in this case, 3 seconds quantized into 12 x 3 = 36 bins), and vertical dimension lists unique samples and notes

3.3 Features

Some of the most powerful aspects of Siren are the features added on top of pattern manipulation mechanisms.

3.3.1 Parameters and Modulations

The pattern functions on Siren can contain any number of *literal*, *temporal*, and *random parameters*. These parameters do not have a specific type, and any kind of input that reconstructs a syntactically correct pattern is accepted (see Fig. 3).

A literal parameter is defined as any substring enclosed in grave accents (`). They can be used in multiple places in the same instance, resulting in the ability to create complex relationships and modulations.

The parameter `` `t` `` is reserved for the temporal parameter and provides the value of timer on execution to the pattern function. It creates connections that behave differently over time and it is especially powerful when used in conjunction with *mathematical expressions*.

Siren allows the user to incorporate random parameters, which are constructed by surrounding two numbers with vertical bars and separated with a comma (e.g. `` `|0,3|` ``). It supports integer and floating point numbers, and could also be combined with mathematical expressions. This has the potential to add an extra level of randomness to the compositions.

3.3.2 Mathematical Expressions

Siren allows using a wide range of mathematical expressions that enhance the computational and algorithmic aspects of pattern creation. These expressions can be employed in any part of the implementation of pattern functions, and upon successful reconstruction, the expression is replaced with its final value. It is notated by surrounding the expression with ampersands (`&`). An example with temporal parameter is:

```
slow &log(`t`, 2)& $ sound "gen1" # end "`t`"
```

where the duration of triggers is in relation with the logarithm of the speed of the pattern. The expressions support numerical spaces, symbolic calculations, trigonometry, vector and matrix arithmetic [11].

3.3.3 Global Modifiers

Another feature of Siren is the *global modifiers* that behave either by prepending transformation functions or appending parameters to the patterns. For example, `#speed -1`

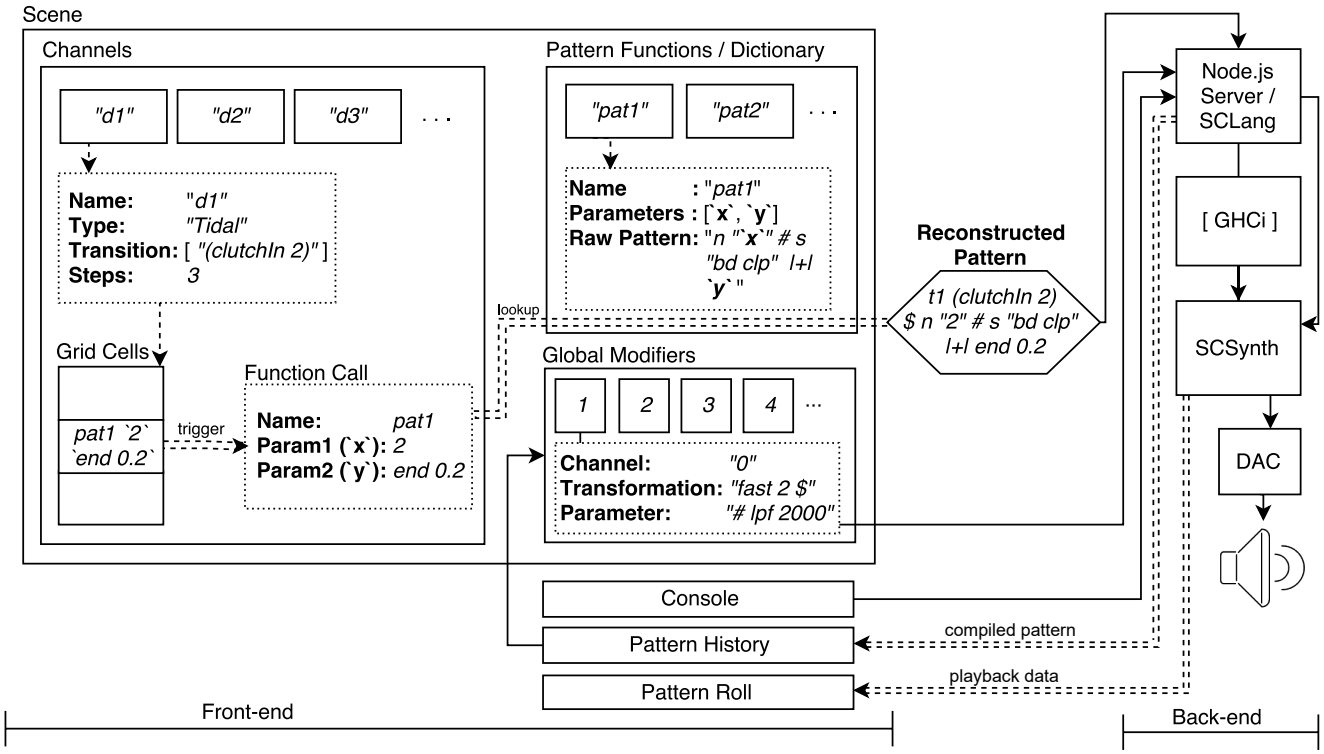

Figure 3: Block Diagram of Siren – On the left, *front-end* lays out the hierarchical structure, and on the right, *back-end* reveals behind-the-scene components. Solid boxes denote prominent modules and their placement reveals the hierarchy relations. Dashed boxes and arrows provide examples to the components. Solid arrows denote message propagation and double-dashed lines represent labeled actions. Square brackets ([ ]) mark optional fields.

reverses current playback, and `#coarse 2` halves the sampling rate. These modifiers could be either applied to all active channels, or a subset of those based on the user input (`'1 2'` affects first two channels where `'0'` is applied on all channels).

Included within this module is a sequencer which can apply the modifiers in specific time intervals. In addition to its dramatic altering power in the musical output, this sequencer supports the main time structure by introducing another mechanism for algorithmic control.

3.3.4 Pattern History

The live-coding paradigm does not account for the past playback data due to its intrinsic liveness. However, when the paradigm is transformed into compositional structure with various parameterizations, reviewing historical data gains importance. This module stores successfully compiled patterns in each channel to ease referring back to the recent playback. However, it currently stores only the most recent playback, but it is trivial to extend its functionality to span entire session.

3.3.5 Pattern Roll

It is possible for algorithms to create patterns otherwise be too laborious to create by hand. However, while these patterns can be interesting, they sometimes lack the precision required in a composition. In theory, it is possible to *tune* a pattern to serve the desired purpose but this may require a profound change in the very core of the algorithm; yet it is probable that tuned pattern may not be as precise as it is desired to be. In step sequencers, this precision is supplied by its step-wise nature where each step is executed individually.

Pattern roll (Fig. 2), inspired by the piano-roll in traditional DAWs, is the visual editing tool of Siren. This module is dedicated to record instances of SuperCollider playback and serves as a visual tool to understand relationships between individual triggers.

This module supports various edit options to the recorded sequence, for example, adding, deleting, or otherwise modifying notes. This allows specific refinements to the patterns where it could be too difficult to establish the desired effect by modifying the original pattern function. Each note in the timeline consists of various parameters emitted from the SuperCollider (e.g. `end, speed, coarse`, etc.), and each parameter value can be edited individually. Subsequently, modifications can be played back synchronously with the main sequencer. The horizontal axis denotes quantized time bins, and vertical lists the names of unique samples and notes. The visibility of labeling on the vertical axis can be toggled to reduce clutter (see Fig. 1 and 2). Default sequence length is 8 seconds and each second is quantized into 12 bins. However, both parameters can be edited using the dedicated textboxes on the interface.

3.4 Implementation Details

The current implementation of the system is built as a JavaScript web application using Node.js [23]. It relies on a number of open-source libraries, notably, React[5] for the construction of the user interface, and CodeMirror[6] for editing text with syntax highlighting for pattern code.

The back-end, which interfaces with GHC[7] and SuperCol-

[5] `https://reactjs.org/`
[6] `https://codemirror.net/`
[7] Glasgow Haskell Compiler

lider, is built using Node.js. This core of the system acts as a bridge between GHC and the Read-Eval-Print-Loop (REPL) class of JavaScript. It is integrated with SuperColliderJS[8] which is a JavaScript library for communicating with and controlling SuperCollider. The back-end starts a terminal that communicates directly with the compiler, and compiles the given Haskell code in the same way as contemporary text editors such as Atom, Emacs and Vim do.

4. COGNITIVE DIMENSIONS

Originally introduced by Green to investigate the psychology of programming languages [9] based on research in cognitive science, and subsequently adapted towards music notation systems and musical user interfaces [6, 20, 18], the *Cognitive Dimensions of Notations* framework stands out as an important criterion for formalized evaluation of Siren. Each dimension is intended to describe a distinct factor related to the usability of a particular notation and interfaces. The dimensions aim to relate to the properties such as *granularity* (considered on a continuous scale from high to low), *orthogonality* (independence from other dimensions), *polarity* (characterizing 'desirability' in a continuous manner for a given context, not necessarily in terms of the 'good' and 'bad'), and *applicability* (in terms of a broad relevance to any kind of notation)" [18]. Due to the real-estate restrictions of this paper, we omit enumerating and explaining each dimension individually but challenge the reader to examine them at their leisure.

Building on Green's definitions, Nash translates the cognitive dimensions into a set of questions that can easily inform the end-users. He investigates these questions in the context of traditional paper scores, Max audio synthesis environment (Max/MSP), and digital audio workstations (DAWs) [18]. Even though it would not be tractable to formulate one-to-one mappings between the features of Siren and specific questions, we propose a personal evaluation of a subset of them below:

Visibility - *"How easy is it to view and find elements or parts of the music during editing?"*

The transparency is one of the key aspects in Siren as it interfaces the *textual* information (i.e. pattern language). Channels lay out the temporal view and pattern roll serves a visualization tool for the current playback that provides greater resolution on how each pattern unfolds over time (see Fig. 2). Just like in DAWs, the distribution of the musical information over the layout may enhance the visibility if related elements are placed and re-sized efficiently.

Juxtaposability - *"How easy is it to compare elements within the music?"*

Among the modules in Siren, the major components that influence musical output are channels and patterns. The possibility to reorder items inside these modules allows side-by-side comparisons of musical elements. As the pattern dictionary expands, it may become harder to find and reorder a pattern of interest; however, the hierarchical scene structure could be used to prevent dictionary clutter.

Hard Mental Operations - *"When writing music, are there difficult things to work out in your head?"*

Textual programming can be considered one-dimensional in the sense that the relationships are encoded only in text. In *visual programming* (i.e. Max/MSP), in addition to the textual input inside the elements, their placements and interconnections also matters. However at the end, since the encoding is compiled into machine instructions regardless of its visual representation, this dimensionality only pertains to the user experience, not to the workings of the process itself [16]. In the case of Siren, the introduction of a temporal structure, which is essentially textual but communicated as a visual component, eases the cognitive load on the user by visualizing the execution order. In addition, pattern roll visualizes the playback and provides another sensory input that ultimately helps the perception of the pattern progression.

Hidden Dependencies - *"How clear are the relationships between related elements in the notation?"*

Siren uses channels and pattern dictionaries to communicate between sequencer calls and actual patterns. This simple communication presents a familiar approach to cross-reference different patterns across the system. In the case of crowded scenes, it may be harder to differentiate between similarly titled elements. However, following a descriptive and distinctive naming system could eventually overcome this issue.

Conciseness / Diffuseness - *"How concise is the notation? What is the balance between detail and overview?"*

Sequencing the elements using the channel grid provides greater visibility on the progression of items in its cells. The size of each step in the grid is virtually unlimited, and thus may require for scrolling and re-scaling to peek at the bigger picture. Subsequently, the pattern roll module provides visual insight into detailed playback data in terms of its timing and properties.

Provisionality - *"Is it possible to sketch things out and play with ideas without being too precise about the exact result?"*

In Siren, every scene can be used as a sketchbook to incubate compositional ideas. As such, the volatile sketching of patterns can be executed very quickly by utilizing the console module that acts as a traditional text editor. Findings in the initial experimentation stage lay out the fundamental ideas, which are subsequently implemented in a more exacting form in the composition.

Secondary Notation - *"How easy is it to make informal notes to capture ideas outside the formal rules of the notation?"*

On the interface of Siren, each module serves a defined purpose and offers fixed tools to manipulate data. In terms of functionality, the system falls short on transforming the notation to an informal and improvisational tool, yet rearranging the layout to meet specific needs yields powerful scenarios for experimentation. Moreover, Siren supports secondary notations through a fusion of textual and visual approaches to the programming. Annotating the programming languages is still possible through descriptive comments on the text body. It provides non-executable and convenient statements to realize quick notes, sketches, and prototyping.

Role Expressiveness - *"Is it easy to see what each part is for, in the overall format of the notation?"*

Siren uses a single view that exposes desirable elements of a composition through a modular layout. Each module has a header that clearly specifies its purpose of the component, however differentiating items at the first glance might be puzzling. The hierarchical structure is not immediately visible on the interface but inherently constructed throughout the system. Also, channel headers uses color to demonstrate different types.

Premature Commitment - *"Do edits have to be performed in a prescribed order, requiring you to plan or think ahead?"*

Creation and manipulation of the composition can be per-

[8]https://crucialfelix.github.io/supercolliderjs/

formed in any order. This flexible workflow allows for musical accidents, which may ultimately turn out to be sonically enjoyable. On the other hand, in conventional step sequencers, the edits need to be planned if the aim is to introduce, for instance, a big breakdown in the middle of the composition. This requires precise adjustments before and after the event. Comparatively, the scene concept in Siren can be utilized to introduce a break or a verse. In a way, it is designed to allow mistakes which may end up fueling inspiration.

Error Proneness - *"How easy is it to make annoying mistakes?"*

The system, in its essence, depends on text input for naming scenes, patterns, parameters and many other aspect of the interface. It is possible to make mistakes in cross-referencing the grid entries and corresponding pattern function names. However, if the function call fails to reconstruct syntactically correct patterns, both the interface and back-end compensates the mistake without disturbing the musical output. Substantial actions, such as completely removing a scene, is guarded with a two-step action requirement: clicking the remove button, and approving the selection on the alert box.

5. CONCLUSION

In this paper, we introduced Siren, an environment that supports algorithmic composition and live-coding through pattern programming. It constructs a carefully designed user-interface on top of the computational pattern creation and features various interface-level properties that, ultimately, empowers users to be articulate in music creation. After discussing the key features of the interface and providing an overview of technical aspects, we analyze the usability of the system in the perspective of the cognitive dimensions [9], and answer a subset of questions that are developed around these concepts [18].

6. ACKNOWLEDGMENTS

Authors would like to thank to creators and contributors of TidalCycles and SuperCollider for their endeavours in creating and actively maintaining these languages. In addition, we also thank Kim Bjørn for including Siren into the successfully crowdfunded book *Push Turn Move: Interface Design for Electronic Music* [4] along with SuperCollider and TidalCycles.

7. REFERENCES

[1] Ableton. Link. `https://github.com/Ableton/link`, 2016.

[2] R. Bell. An approach to live algorithmic composition using conductive. *Proceedings of LAC 2013*, 2013.

[3] R. Bell. Experimenting with a generalized rhythmic density function for live coding. In *Linux Audio Conference*, 2014.

[4] K. Bjørn. *PUSH TURN MOVE: Interface Design for Electronic Music*. Kim Bjørn, Vanløse, Denmark, 2017.

[5] A. Blackwell and N. Collins. The programming language as a musical instrument. *Proceedings of PPIG05 (Psychology of Programming Interest Group)*, 3:284–289, 2005.

[6] A. F. Blackwell, T. R. Green, and D. J. Nunn. Cognitive dimensions and musical notation systems. In *Proceedings of International Computer Music Conference, Berlin*, 2000.

[7] L. Church, C. Nash, and A. F. Blackwell. Liveness in notation use: From music to programming. In *Proceedings of the 22nd Annual Workshop of the Psychology of Programming Interest Group (PPIG 2010)*, pages 2–11, 2010.

[8] K. Collins. *Game sound: an introduction to the history, theory, and practice of video game music and sound design*. Mit Press, 2008.

[9] T. R. G. Green and M. Petre. Usability analysis of visual programming environments: a 'cognitive dimensions' framework. *Journal of Visual Languages and Computing*, 7:131–174, 1996.

[10] C. Ince and M. Toka. Siren: Hierarchical composition interface. In *Proceedings of International Computer Music Conference, Shanghai*, 2017.

[11] J. Jong. *Math.js*. `https://github.com/josdejong/mathjs`, 2018. Accessed: 2017-03-20.

[12] R. Kirkbride. Foxdot: Live coding with python and supercollider. In *Proceedings of the International Conference on Live Interfaces*, 2016.

[13] A. McLean. The textural x. *Proceedings of xCoAx2013: Computation Communication Aesthetics and X*, pages 81–88, 2013.

[14] A. McLean. Making programming languages to dance to: live coding with tidal. In *Proceedings of the 2nd ACM SIGPLAN international workshop on Functional art, music, modeling & design*, pages 63–70. ACM, 2014.

[15] A. McLean and G. Wiggins. Tidal–pattern language for the live coding of music. In *Proceedings of the 7th sound and music computing conference*, 2010.

[16] C. A. McLean et al. *Artist-programmers and programming languages for the arts*. PhD thesis, Goldsmiths, University of London, 2011.

[17] J. Nakamura and M. Csikszentmihalyi. The concept of flow. In *Flow and the foundations of positive psychology*, pages 239–263. Springer, 2014.

[18] C. Nash. The cognitive dimensions of music notations. In *The Cognitive Dimensions of Music Notations*, 2015.

[19] C. Nash and A. Blackwell. Tracking virtuosity and flow in computer music. In *Proc. ICMC*, 2011.

[20] C. Nash and A. Blackwell. Liveness and flow in notation use. In *Proc. NIME*, 2012.

[21] C. Nash and A. Blackwell. Flow of creative interaction with digital music notations. In K. Collins, B. Kapralos, and H. Tessler, editors, *The Oxford Handbook of Interactive Audio*, pages 387–404. Oxford University Press, New York, 2014.

[22] S. L. Tanimoto. Viva: A visual language for image processing. *Journal of Visual Languages & Computing*, 1(2):127–139, 1990.

[23] S. Tilkov and S. Vinoski. Node.js: Using javascript to build high-performance network programs. *IEEE Internet Computing*, 14(6):80–83, Nov 2010.