Siren: Hierarchical Composition Interface

Can Ince
University of Huddersfield
can.ince@hud.ac.uk

Mert Toka
University of California, Santa Barbara
merttoka@mat.ucsb.edu

ABSTRACT

This paper introduces Siren*, a software environment that aims to reinforce ties between the live-coding performance and algorithmic composition. It is based on a hierarchical structure, which propagates modifications to the lower levels with minimal effort, and a tracker-inspired user interface, which allows sequencing patterns on broader timelines. It is built on top of the TidalCycles language [1, 2], and the use of functional programming paradigm allows uninterrupted audio output even the syntax does not match the required format. In addition to the pattern composition,* Siren *supports programming variations of and transitions between patterns. It features polyrhythmic timers, pattern history, local and global parameters, and mathematical expressiveness. Apart from its musical opportunities, the interface leverages a handful of highlights such as user authentication and import/export functionality.*

1. INTRODUCTION

The practice of live-coding music has grown dramatically over the recent years, with the increasing number of practitioners being supported by a growing number of programming languages and environments [3]. Meanwhile, hybrid systems for algorithmic composition that combine multiple approaches led to new possibilities for expression in live-coding [4]. These systems make use of two or more languages and communicate through various bindings [5]. The principal drawback of such hybrid systems is that they may be very complicated, which proposes a high learning curve for users and, therefore, hinders accessibility. However, hybrid systems can also offer novel ”powers” to composers and performers by bringing different capabilities together.

Siren [1] implements a tracker-inspired user interface that acts as a wrapper on TidalCycles musical pattern programming language [1, 2] by layering a terse parser for the deconstruction and reconstruction of patterns. The system's grid-based user interface allows for the composition and sequencing of multiple patterns.

[1] The word *Siren* denotes a loud, attention-grabbing noise, as well as being the name of a dangerous creature that lures and captures sailors with her magical voice and music in the Greek mythology.

2. BACKGROUND AND PREVIOUS WORK

Below, we present an overview of the works that have inspired and informed the design of *Siren*. These works can be categorized into two broad areas: programming languages for composing patterns and tracker-based applications.

2.1 Musical Pattern Languages

Programming has started to become an inspiring medium for making music and it has been advanced by several programming languages. An early example is SuperCollider [6], a platform for audio synthesis and algorithmic composition, most notably the `Pbind` class has extensive capabilities for pattern programming. More recently, Conductive [7] offered a higher-level abstraction for generating sets of greater or lesser densities based on an initial pattern and storing them in a table indexed by the level of density [8]. Finally, TidalCycles (Tidal, in short) [1, 2] introduced an embedded Domain Specific Language (eDSL) for composing patterns as higher order structures with a highly economical syntax. Both Conductive and Tidal allow the use of Glasgow Haskell Compiler (GHC) to trigger a synth using the Open Sound Control (OSC) [9] protocol [3].

Although *Siren* is not a programming language by itself, its creation has been highly inspired by these efforts. The main motivation for the system emerged from the lack of features in functional programming and Tidal for creating compositions, such as memory, parameters, and abstractions. The system is built on a back-end structure for patterns to achieve a fusion between the affordance of Tidal and Conductive, while allowing both to be integrated as the main pattern language. To create polyrhythmic timers for each channel, we borrowed the ”TempoClock” concept from Conductive.

2.2 Music Trackers

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 [10] in 1987, NoiseTracker [11] in 1989, and Protracker [12] in 1990. The tracker user interface depicts compositions as rows of discrete musical events positioned in columnar channels [13]. Each cell in a channel can hold a note, parameter change, 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. While early trackers had only rudimentary sampling capabilities, later implementations have added synthesis, MIDI input/output, plug-in

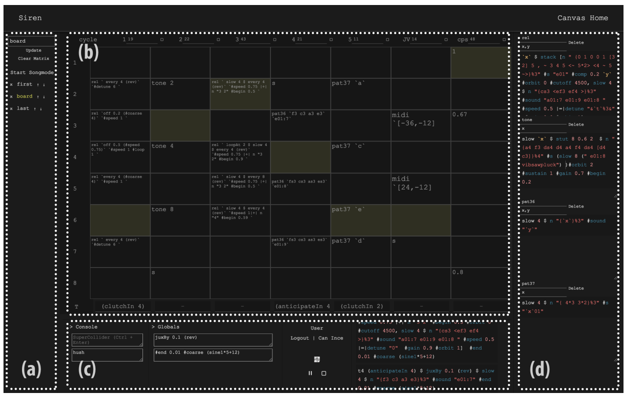


Figure 1. Interface of *Siren*: (a) scenes, (b) grid with instances, (c) console, menu, pattern history (left-to-right), and (d) pattern dictionary

hosting, and recording functionality (e.g. Renoise [14]).

3. CONCEPTUAL STRUCTURE

Below, we explain ideas and working principles behind *Siren* (Figure 3). These can be organized into three categories: patterns as functions, redefining the tracker, and hierarchical composition.

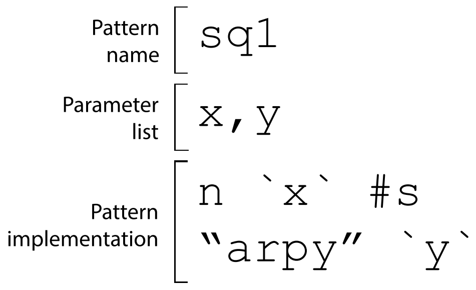
3.1 Patterns as Functions

In *Siren*, the patterns correspond to functions that are referenced from the grid cells. Each pattern comprises function calls, which can be written inside the cells on the tracker grid. Each cell can accept a single function call. The implementations of these functions are written in the *dictionaries* that are on the right side of the interface. Function calls, as in many programming languages, are constructed by the name of the function and its optional parameters.

Siren is designed to treat stored patterns as functions. The *instance* (e.g. Figure 4), which is written in one of the cells in the grid, and the *pattern function* (e.g. Figure 2), which contains the implementation of the instance and located in the dictionary. Furthermore, instance has two components, function name and parameters that are enclosed in grave accents (`). Grid-oriented interface keeps track of the state of each cell in the scene. Once the timer of the cell is active, it looks up for the pattern specified by the name and parameters.

These pattern functions can be called from any cell of the grid. When the timer triggers the related cell, a look-up in the pattern dictionary is performed. Once the desired pattern is found in the dictionary, the pattern is reconstructed by parsing its parameters with the regular expressions and replacing them with the user input. If the related pattern in the dictionary is reconstructed correctly, GHC compiles the pattern to be received as an OSC message by SuperCollider.

Treating patterns as functions enables a new level of abstraction by allowing a structural approach to Tidal and by introducing parameters that can be used in any part of the pattern (see Section 4.2). The timer of each channel determines the temporal parameter, which is the time value of the corresponding channel, and this parameter can be accessed within the patterns. Once the pattern function gets the required parameters, it gets reconstructed by the parser



```
sq1
x,y
n `x` #s
"arpy" `y`
```

Figure 2. A pattern function in the dictionary

to fit the required syntax. In that sense, functional structure supplies memory to Tidal.

3.2 Redefining the Tracker

In a conventional tracker, sounds are triggered by note or control change messages that are specified in each cell. Conversely, in *Siren*, each cell specifies a tidal pattern function call. This, combined with the introduction of cyclical structures from Tidal, brings new capabilities to the tracker concept.

Using a functional programming paradigm was a practical choice as the nature of the trackers with note-on/off system was taken into consideration. *Siren* differs from other music tracker interfaces mainly by introducing patterns functions instead of notes or control messages in each cell and having an individual timer for each channel. This opens up diverse expressive capabilities for the performer/composer such as using global parameters in a performative fashion or randomizing the transitions with controlled boundaries.

3.3 Hierarchical Composition

The creation of a pattern in *Siren* introduces a rhythmic element or a cycle, yet the pattern itself lies within its temporal bounds and linearity. Sequencing the code allows the performer/composer to break out of linearity by modifying the cycle with different variables and transitions while allowing events remain coherent in their single and linear clock cycle. This mechanism can be exploited to expand a sense of repetition and structure on a larger scale. In the system, this expansion can be achieved using several hierarchical structures.

The main structure of *Siren* is the concept of the *scene*, which acts as a top-level container for other features. Each scene accommodates a unique *grid* (Figure 1b), where the columns correspond to the various *channels* and the rows determine the temporal bounds of the scene. A scene also has a *dictionary* (Figure 1d) for the storage of *pattern functions* that have the ability to define their own parameters and implementations (Figure 2). Instances of these pattern functions (Figure 4) can be written into the grid to call corresponding pattern function on timer trigger. In addition to storing pattern functions in its dictionary and the grid cells in its matrix, a scene also has a *duration* array and a *transition* array. Their sizes are bound to the number of

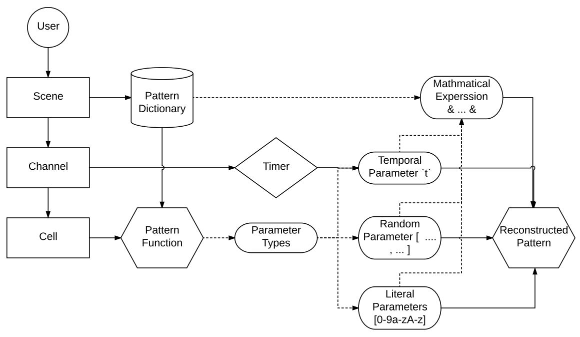


Figure 3. Conceptual Structure of *Siren*

channels, in other words, the number of columns in the grid (see Sections 4.2.4 and 4.2.2).

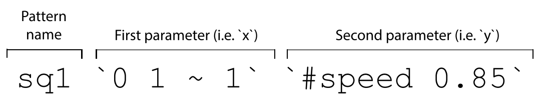
One account in the system can create multiple scenes (Figure 1a), each of which has unique pattern functions and sequences. The construction of the scenes in this manner enables the performer/composer to quickly realize ideas for complex musical structures.

4. SYSTEM STRUCTURE

This section explains the underlying structure of *Siren* system and the workflow for composing with it.

4.1 Overview

The core of the system acts as a bridge between GHC and the Read-Eval-Print-Loop (REPL) class of JavaScript. 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. However, the interactive user interface in *Siren*, in lieu of a mere text editor, allows for the deconstruction and reconstruction of the Haskell code in unusual ways as it is explained in Section 3.



```
sq1 `0 1 ~ 1` `#speed 0.85`
```

Figure 4. An instance of a pattern function in one of the grid cells

4.2 Features

Some of the most powerful aspects of *Siren* are the features that are added on top of Tidal's compositional capabilities (Figure 3). The system also supports networked coding performances and compositions. To join a networked performance, performers can launch the system in their browsers and log-in to a shared account.

4.2.1 Parameters

Each pattern in *Siren*'s dictionary can consist of any kind of Tidal patterns and any number of literal or random parameters. A literal can be used in multiple places in a pattern, hence it can create complex relations when used with mathematical expressions. These parameters do not have a specific type, and any kind of input which allows patterns to be constructed in a modular fashion can be accepted.

Siren also introduces random parameters, which can be constrained within given boundaries. This can add an element of randomness to compositions. This aspect of Siren could be particularly interesting when used to create multi-level randomness inside a pattern (e.g. `irand('x')` where x is the value of the random parameter).

4.2.2 Polyrhythmic Timers and Temporal Parameter

Polyrhythmic events are possible through the implementation of multithreaded timers. Channels have unique cycle durations, and by adjusting the durations independently, complex events can be created. The timer can be manually started, triggered, and stopped. The timer has a temporal parameter, which provides a continuous value to the pattern functions.

4.2.3 Mathematical Expressions

The system allows utilizing a wide range of mathematical expressions that enhances the computational and algorithmic aspect of pattern creation. These expressions can be employed in various places of the implementation of pattern functions, and upon successful evaluation, the final value is replaced with the expression. It is parsed by surrounding the expression with ampersands (&). Integration with parameters, especially constantly increasing temporal parameter, opens up new possibilities for modulating certain parts of the pattern. The expressions support numerical spaces, symbolic calculations, function creations, trigonometry, vector and matrix arithmetic [15].

4.2.4 Transitions

Borrowing the transition functions from Tidal, in *Siren*, it is possible to specify a different transition function for each channel and save them in the scene. This can introduce a unique complexity between patterns.

4.2.5 Global Transformers and Parameters

In *Siren*, we introduce global transformations and parameters that act on all of the channels, by either prepending transformation functions or appending parameters to the patterns. The values of global transformations and parameters change in real-time as the user updates related field on the console. Global modifiers account for dramatic changes in the musical output when used with transformers. For example, `#speed -1` reverses current playback on all channels, `#coarse 2` halves the sampling rate of all active channels, and so on.

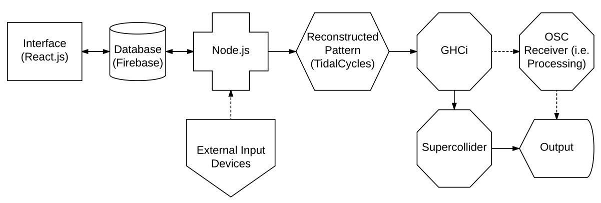


Figure 5. System Structure of *Siren*

5. IMPLEMENTATION DETAILS

Siren is a JavaScript-based web application. The back-end, which interfaces with GHC, is built using Node.js [16]. The React.js [17] library was chosen to build the user interface, due to its stability and active community.

For writing musical patterns, Tidal emerged as the most convenient choice.The use of functional programming paradigm offers uninterrupted audio stream even if the syntax does not match the required format. Due to its nature, Tidal treats compilation as an evaluation of mathematical functions and avoids changing-state and mutable data. Therefore it is highly practical for pattern programming.

The dictionary in *Siren* was implemented using Firebase, which provides a stable NoSQL database. To highlight pattern code, *Siren* utilizes the open-source editor CodeMirror. The editor affords various features such as syntax highlighting and customized themes.

6. FUTURE WORK

We aim to extend *Siren*'s capabilities so that it can become an assistant to the performer/ composer by generating patterns and collaborating on a piece. For pattern generation, we would like to incorporate generative learning algorithms that will, in time, assume unique stylistic identities shaped by the user's actions.

7. CONCLUSION

Siren is a powerful tool to quickly realize musical ideas. It enhances Tidal's pattern capabilities and shifts its purposes towards composition. Having a tracker-inspired user interface on top of the language allows pre-processing of input in myriad ways and works towards an expressive control structure.

An early version of *Siren* has been debuted in the Algorave [2] 5th-year anniversary stream. Our observations showed that the community was excited about the project and they requested for a release. Currently, it resides in its GitHub repository [3] as an open-source project. As of today, the system is still under development and last released version is *0.1.1-beta*. The project has also been featured in TOPLAP [4], the home of live-coding. It has attracted interest from musicians who previously needed to compromise between Tidal expressiveness and sequencing approach.

The most praised aspect of *Siren* is often its versatility; it has many features that distinguish itself from the current live-coding medium, yet it is possible to perform a live-coding session in *Siren* by approaching it merely as an online text editor.

Acknowledgments

Authors thank Alex McLean and other contributors of TidalCycles for creating and maintaining this expressive pattern language. Thanks also goes to Alexander J. Harker for his research support and valuable suggestions.

[2] algorave.com
[3] github.com/cannc4/Siren
[4] toplap.org/siren

8. REFERENCES

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

[2] ——, "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*. ACM, 2014, pp. 63–70.

[3] 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.

[4] A. McLean and G. A. Wiggins, "Texture: Visual notation for live coding of pattern," in *ICMC*, 2011.

[5] G. Papadopoulos and G. Wiggins, "AI methods for algorithmic composition: A survey, a critical view and future prospects," in *AISB Symposium on Musical Creativity*. Edinburgh, UK, 1999, pp. 110–117.

[6] J. McCartney, "Rethinking the computer music language: SuperCollider," *Computer Music Journal*, vol. 26, no. 4, pp. 61–68, 2002.

[7] R. Bell, "An Approach to Live Algorithmic Composition using Conductive," *Proceedings of LAC 2013*, 2013.

[8] ——, "Experimenting with a Generalized Rhythmic Density Function for Live Coding," in *Linux Audio Conference*, 2014.

[9] M. Wright, A. Freed *et al.*, "Open SoundControl: A New Protocol for Communicating with Sound Synthesizers." in *ICMC*, 1997.

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

[11] mandarin, *NoiseTracker V2.0 by Mahoney Kaktus*. http://www.pouet.net/prod.php?which=13360, accessed: 2017-03-30.

[12] eightbitbubsy, *ProTracker*. https://sourceforge.net/projects/protracker/, accessed: 2017-03-30.

[13] M. Gallagher, *The music tech dictionary: a glossary of audio-related terms and technologies*. Nelson Education, 2014.

[14] Renoise, *Renoise*. https://www.renoise.com/, accessed: 2017-03-30.

[15] J. Jong, *Math.js*. https://github.com/josdejong/mathjs, accessed: 2017-03-20.

[16] N. Foundation, *Node.js*. http://nodejs.org/, accessed: 2017-03-30.

[17] F. Inc., *React*. https://facebook.github.io/react/, accessed: 2017-03-30.