Programming For Music: Explorations in Abstraction

Can Ince

School of Music, Media and Humanities

University of Huddersfield

A thesis submitted to the University of Huddersfield in partial fulfilment of the requirements for the degree of

Master of Arts in Music

November 2018

Acknowledgements

I would like to thank several people who contributed to this research, either directly or indirectly. I must first thank Alexander J. Harker for his valuable supervision and mentorship throughout this research. Besides my advisor, I would like to thank to Frederic Dufeu and the rest of the Creative Coding Lab for their insightful comments and encouragement. I also thank to contributors of TidalCycles and SuperCollider for creating and actively maintaining these languages. I thank my friend Mert Toka for the stimulating discussions and his contributions on this research. Last but not the least, I would like to thank my parents and my friends for supporting me spiritually throughout this research and my life in general.

Copyright Statement

Abstract

Creatively, algorithmic processes open up all sorts of possibilities which would be either impossible or too laborious to create by hand. This thesis is my attempt to explore the depth of the rhythm and time through algorithms accompanied by a software, *Siren*, which is designed for pattern sequencing.

This thesis documents an effort in attempting to develop a *novel technical approach to musical composition* that functions not just as a tool, but also as *an extended cognition that shapes the creative process*.

To this end, several ideas and design approaches derived from previous work in computer science, philosophy, music, and other disciplines are utilised to conceive of (and subsequently implement as a software application) a musical interface that is tailored towards algorithmic approaches to music composition. This thesis presents the result of that effort as well as the process of its creation. A discussion evaluating the abstractions and cognitive dimensions which inform the design and implementation of the application is also included.

Beside basic curiosity and experimentalism, there are several reasons why I wanted to adopt algorithmic methods, and this thesis will serve as a guide and a notebook towards achieving stability in music with the fusion of various concepts.

Keywords: digital music notation, algorithmic composition, abstraction, trackers, code scores, patterns

Contents

List of Submitted Materials

1. Siren - Algorithmic Composition Interface, including the session files
2. Raw-Cuts - Selected excerpts from live recordings
3. Edited-Tracks - Edited and post-processed tracks
4. Pattern studies - Earlier studies with Siren
5. Screencasts of Siren sessions and tutorials
6. ICMC17 and NIME18 papers

List of Figures

List of Tables

Chapter 1

Introduction

This chapter provides a foundation based on previous work and situates the thesis in the context of related research. To this end, six topics are discussed. The first is notion of *algorithms*, with emphasis on how they find their place in musical composition. The second is the notion of *abstractions* as they apply to the information theory, and how this theoretical concept underpins ideas that manifest in musical interfaces in general, and *Siren* in particular. The subsequent section overviews the concept of *creative coding*, with emphasis on "live coding" of music. Following this is a discussion on the conceptualisations of *time* and *patterns* in music, and how different abstract understandings of time and patterns inform approaches to music composition, especially with regard to digital musical instruments. Finally, this chapter discusses the impact of *user interfaces* on musical creation, paying special attention to music trackers.

1.1 Context

1.1.1 Motivation

Live coding is the act of using a programming language as the primary instrument for musical expression. However, most of the live coding languages are tailored for performance rather than composition. The focus on this thesis will be directed towards *TidalCycles (Tidal)* which is an *Haskell* based live coding language that depicts time in *cycles*. While it is possible to represent long-term structure in *Tidal*, it is mainly built for live coding performances where the long-term structure is controlled and continuously manipulated by the performer [40]. This absence of a facility for defining a long-term structure in *Tidal* is one of the key issues that has driven the creation of the software described in this thesis.

The creation of Siren, the system described in this thesis, was motivated by the aforementioned issue, and also by my own creative needs as a live coder. My own approach to music making builds on the idea of working in layers and being able to switch between multiple elements instantaneously.

1.1.2 Research Aims

The above motivation has guided the research project presented in this thesis, where the following has been accomplished:

1. Building a musical interface to be used within the live coding context which primarily builds on TidalCycles; both due to its technical provisions, providing means to store and structure patterns in a musical fashion.

2. Understanding design characteristics that are relevant for a software application that aims to function as an extended cognition facility, specifically for supporting a hybrid approach to live coding and composition.

3. Using the aforementioned approaches in musical compositions.

1.1.3 Clarifications

1.1.3.1 Nature of Collaboration on the software

Initially, I started this projects in the third month of my research degree simply as a need in my musical workflow. The following components of the software were originated by myself and present in the first demo version of the software; the compiler integration, the tracker and modular pattern concept, the boot schema, scenes, history, and global controls. After the initial prototype, Mert Toka[1] contributed with some performance and cosmetic related improvements such as dynamic layouts and syntax highlighting. After the initial International Computer Music Conference 2017 publication [25], while my main focus was on music making and instrument design, he focused on testing different graphics libraries for the pattern visualiser for future improvements and features. A git commit history is also included in the submission to provide more information on the progression of the software.

[1] https://github.com/merttoka

1.1.3.2 Submitted Materials

Tracks that are included within the submission are made by using Siren, in conjunction with *TidalCycles* [40] as the main sequencer, the *SuperDirt* sampler [1] from *SuperCollider* [34] and a *Nord Modular G2 Engine(G2)* [8] which runs three different digital signal processing(DSP) patches per performance[2]; a drum machine, a bass, and a voice patch. On top of G2's sound sources, there are also *SuperCollider* sources which get mixed with the G2 before going out to the loud-speakers. The consideration in choosing G2 was based on its mobility of the hardware and flexibility of its workings.

In the submitted "audio" folder, there are three folders namely, "Raw-cuts", along with "Edited-tracks" and "Pattern-studies". "Raw-cuts" are the selected audio excerpts from the recordings made during the research period. These excerpts include improvisation with Siren scenes with modulation tweaks compiled on-the-fly and recorded. Secondly, "Edited-tracks" are the edited and post-produced version of some of the patterns from Raw-cuts. Post-production includes additional mixing and layering. Lastly, "Pattern-studies", are the selected recordings made with an earlier version of Siren and included to give the reader insight into the progression of musical output. An earlier recording of a radio performance using Siren is also located within this folder. Moreover, The folder named "Screencasts" contains the screen-captures which showcase the software. These videos contain various bits of the recordings in the "Raw-cuts" folder. The final version of the software, as well as the session files, are also included within the submission.

1.2 Algorithms in Music Composition

The Greek philosopher, mathematician, and music theorist Pythagoras (ca. 500 B.C.) documented the relationship between music and mathematics that laid the foundation for our modern study of music theory and acoustics. Greeks believed that the understanding of numbers was key to understanding the universe. Although there are numerous treatises on music theory dating from Greek antiquity, the Greeks left no explicit evidence of how mathematical procedures applies to the composition of music [55].

Since then, several categories have emerged from the study of algorithms for music composition, including the aleatoric (or chance) methods (e.g. Cage), determinacy

[2]The performance concept in Nord Modular refers to an higher level container which holds four different DSP patches

(e.g. Schoenberg, Webern, and Berg), and stochastic (or probabilistic) methods (e.g. Xenakis and Hiller). Composers have also been applying not only mathematical models, but also biological paradigms such as L-systems [61] and genetic algorithms [21] to the creation of music. These days, since aforementioned processes can be modeled computationally, almost all of these models may be used for music composition in computational environments [55].

Before we move on to other topics, this chapter will benefit from a more involved discussion of what an algorithm and composition is *is*.

An *algorithm* is defined as "a fixed step by step procedure for accomplishing a given result [...]" and "a defined process or set of rules that leads to and asserts development of a desired output from a given input" in the Computer Dictionary and Handbook [56]. However, the precise meaning of the term *algorithmic* often depends on the context it is used in. Several criteria have been proposed to form a generic understanding of the term. These include the requirement to have a finite number of steps, to have both well-defined input(s) to and output(s) from the algorithm, to yield a result in a finite period of time, and to have a precise definition for each step of the algorithm [55]. If the algorithm is to be programmed to function as a human being would, it can be said to involve decision-making and development procedures. In this case, the terms *random*, *stochastic*, or *aleatory* (all pertaining to chance) would more accurately describe a process, rather than *algorithmic* (i.e. using defined logical procedures) or *intelligent* (simulating human mental processes) [57].

The meaning of the term *composition* relates to putting together parts into an unified whole. Moreover, the process of composing music is often characterised by trial and error. The composer executes an idea, listens, and determines if revisions are necessary. As such, the composer continuously evaluates the effectiveness of a part in relation to the whole.

In my work, I conceptualise a *compositional algorithm* as a set of rules, a description and transformations in an artificial computer language of a pattern creation process. It can be understood as a written *score* that embodies a procedure for composing. However, instead of human instrumentalists playing the music described in the score, a computer does.

An early example of algorithmic music is the works of Iannis Xennakis. Xennakis generated his first piece Metastaseis (1955-56) [53] dealing with large number of data sets, and started using computers as a necessity to assist these calculations, producing three works in 1962 [64]. Since then, numerous tools have been developed for composers looking to explore such ideas. Max/MSP [51], a *modulable* programming

environment, SuperCollider [34, 35], a platform for audio synthesis and algorithmic composition, and many others has been used by composers and researchers who have been inspired by the stochastic and dynamic algorithmic methods. Such ideas and theories can be re-modeled efficiently in these new environments and to date, there have been developments in the field through a multitude of algorithmic composition tools, which allow a composer to work more quickly by offering them a close match between the creative methodology and the algorithmic implementation. A more recent manifestation of this approach lies at the heart of the *Algorave* movement, which is centred around live-coded music, often accompanied with live-coded visuals. The audience experience in these performances is quite rich, and accounts of Algorave events report audiences "looking up at the projected codes rather than each other", and that it feels "in many ways like an art performance with some dancing" [2].

1.3 Abstraction

The concept of *abstraction* is central in computer science [60]. An abstraction can be defined as "a concept or idea not associated with any specific instance" or "the process of formulating general concepts by abstracting common properties of instances" [44]. It is notable that the root of the word, *abstract*, relates to the immaterial and vague. However, abstractions in computer science don't always have to be so, as they may represent particular domain-specific notions rather clearly.

The Free Online Dictionary of Computing defines abstraction as "[producing] a more defined version of some object by replacing variables with values (or other variables)". According to Dijkstra, abstraction in computer science is not as much about ignoring details as capturing essential details. Abstraction is the tool that gives computer science its algorithms and variables, and it "permeates the whole subject [of computer science]" [12].

The instantiation of abstractions is often phrased as software reuse, as described by Krueger [29]:

> Abstraction plays a central role in software reuse. Concise and expressive abstractions are essential if software artefacts are to be effectively reused. Effectiveness of a reuse technique can be evaluated in terms of cognitive distance and an intuitive gauge of the intellectual effort required to use the technique. Cognitive distance is reduced in two ways: (l) Higher level abstractions in a reuse technique reduce the effort required to go from the initial concept of a software system to representations in the

> reuse technique, and (2) automation reduces the effort required to go from abstractions in a reuse technique to an executable implementation.

This idea can also be applied to practice, where the abstractions are essential in order to add dynamism to the sound. A written code can be propagated in different musical forms with a slight changes in the input. Here, the instantiations of abstractions may be considered as affordances for virtuosity; since a bi-directional relationship is achieved between the output realised by the system and the inputs provided by the musician (see Sec.1.4).

A stable structure for the usage of abstraction is realised with connecting the nodes. As we go higher on the abstraction level/order, the parameters that controls the abstraction is reduced. Even if the input may seem unwild, it may cause musical output to become overly dynamic. One particular case of how such an understanding of abstractions informs the design of Siren is the notion that Patterns (see Sec.1.6). In Siren, patterns are represented in a modular fashion. This modularity is supplied by the pattern functions (see Sec.1.3) presented within the software, which allow for the instantiation of an abstracted patterns.

1.4 Creative Coding and Live Coding

In my own practice, *programming* is used consistently as a way of musical thinking. It would be fair to state that the development of a musical ecosystem involves many *creativity components* which, ultimately, enables progress within the creative process. The word *programmer* is often used to implicitly refer to a kind of technician who tends to a computing machine. However, the same word can be used beyond this context, in relation to a *craft* which is situated in an artistic context. Confronting the singular identity of the programmer as an artist is a particularly salient excerpt from John Stuart Mill [43] which was also mentioned by Donald Knuth:

> Several sciences are often necessary to form the groundwork of a single art. Such is the complication of human affairs, that to enable one thing to be done, it is often requisite to know the nature and properties of many things. Art in general consists of the truths of Science, arranged in the most convenient order for practice, instead of the order which is the most convenient for thought. Science groups and arranges its truths so as to enable us to take in at one view as much as possible of the general order of the universe. Art brings together from parts of the field of science most

> remote from one another, the truths relating to the production of the different and heterogeneous conditions necessary to each effect which the exigencies of practical life require .

One of the key issues in the practice of creative coding is the particular amount of *time* dedicated to working on a potential implementation without yet obtaining a result. However, it can be said that there exists a potential for *enlightenment* when working with yet-non-functional (or even malfunctioning) code. In this state, it is possible for the programmer to conceive of a potential feature where the ideas can be emergent. A decision made at this point, affects the flow of the software and eventually reflects on the musical output. Thus, in the process of developing an instrument for creative coding, one goal is to expose the programmer to such windows and perspectives. Even aspects of ordinary life can be exploited to serve the creative process in such ways, with the appropriate tools.

Live coding is a solid examples of *creative coding*. It is an often improvised performance practice which includes the writing and/or editing of a generative rule set that governs a *current* and ephemeral flow of artistic creation. As Magnusson writes, "The default mode of live coding performance is improvisation". It is an act of expression and communication that involves the creation, modification, and display (as-is or as manifest in their output) of codes. Regardless of the "superficial" appearance of the live coding act, many fundamental aspects of live coding are shared with other, more conventional artistic practices. The practice of live coding involves a broad range of variation and style. For example, some performances may be based on the previously written code, while others may be coded from scratch. The feedback from the interpreter which can become a part of the performance – in some performances this feedback is sparse, or not shared with the audience, while in others it is verbose.

The history of the live coding technically started with the implementation of first *interpreted programming language*, LISP. In the mid-1980s, there were some activities involving live coding but it began to gather attention in the early 2000's with *Slub*'s first live coding performance using a custom software made with LISP[11]. After a year later, one of the first environment dedicated to live coding, *JITLib* library for SuperCollider 2, was published by *Julian Rohrhuber* and followed by many others [52]. Live coding is a good example to computer programming retaining aspects of a science and an art, and that the two facets complement each other beautifully[28]. These efforts are mainly geared at providing tools for live performance. In addition to supporting performative live coding, the effort described in this thesis is meant to

provide the means for musicians to utilise live coding for premeditated compositions where patterns are arranged towards a particular long term structure.

1.5 Time

In this section, I will briefly explain the ideas that shaped my approach to pattern creation and sequencing.

Rhythm, meter and duration are the concepts that govern the human understanding of musical temporality. Yet, a precise characterisation of *time* and temporality as we experience it, as it relates to the evaluation of algorithms on a digital circuit, has often been purposefully ignored in computer science [39]. One of the main focus of this thesis is the integration and exploration of temporal structures in algorithmic music (see Sec.2.1). To this end, different methods are used to work with time and conceptualise it. In turn, each of these different methods had a significant impact on the software and compositions.

1.5.1 Rhythm

The concepts of *repetition* and *variation* (of patterns, in time), which can manifest in balance or in a way in which one overpowers the other, supports to enable the experiences of *similarities* that arise from *repetition*, and *progression* or *generation* which can serve to both affirm or negate *anticipation* at various points in time [30]. This research focuses mostly on two rhythmic techniques namely *isorhythms* and *polyrhythms*. Isorhythms involve simultaneous overlapping rhythmic structures that are different in length, whereas polyrhythms involve concurrent overlapping of rhythmic structures that are divided differently. In other words, the isorhythms manifests themselves through phasing effects while polyrhythms are the kind of 3 against 4 or other syncopated forms of rhythm. Specifically *polyrhythms* and their involvement in the author's musical output will be further explored in the third chapter.

In the musical interface that is proposed in this thesis, the articulation of such effects are supported extensively through different timer structures and parameterizations that allow concurrent execution of patterns with respect to different temporal intervals. Hence, it becomes possible for the composer to fluidly transition between different experiences of repetition, while retaining the pattern affordances of the *live coding* and *tracker* paradigms[3].

[3]The reader will find, in the following sections of this chapter, discussions on how significant attention to *patterns* informs the musical composition interfaces, the practice of live coding, and the

1.5.2 Repetition

One important aspect to consider when reasoning about repetition in this context is the *information differential*, i.e. the density of new information over time, and the capability of a human listener to absorb and comprehend (or apprehend) it. Much of the expressive power in musical composition relates to the manipulation of the balance between redundancy and new information. Many powerful musical forms such as the canon, the fugue, the sonata, and the rondo have been shaped around repetition as a core value [57]. Repetition and continuity are potent tools in the composer's arsenal, which can be utilised to manipulate the *predictability* in musical expression and to shape the ephemeral listener's hypothesis. Both *cyclic repetition* and the *linear repetition* can be part of this mixture and they interfere with one and another constantly [30]. The linear relates to the the monotony of sounds, of gestures, and of imposed structures; while cyclical repetition is periodic, restarting continuously.

In this thesis, and later in the recordings, the concept of the *meter* is considered to be one of the most important aspects of patterns. This is a well established phenomenon that allows us to analyse the characteristics of *rhythm* as discussed in the previous section.

It is important to be able to distinguish between the concepts of *rhythm* and meter. *Rhythm* relates to the occurrence of temporal patterns that are "phenomenally present in the music". Such patterns can be referred to as "rhythmic groups". Temporal patterns, however, are not always strictly founded on the "action duration" of musical event. A rhythmic pattern can occur "between the the start-end points of successive events" [31]. In contrast, meter involves the initial perception of the listener, as well as subsequent anticipation of a series of events that the listener abstracts from the rhythmic surface of the music as it unfolds over time. Altering the meter of patterns manifests and manipulates flow in the musical structure, and thereby allows for emotion to be encoded and decomposed by the transitions and differentials between musical information.

1.6 Pattern Programming

In music, by applying algebraic operations to temporal structures, such "mathematical models of calculation and measure" can be represented as *events in time*. These

affordances of musical trackers.

temporal structures can be combined or modified to create sequential events, on a computer. Such ideas are the basis for several programming languages for the purpose of specifying musical patterns.

In SuperCollider, one of the most notable features for pattern programming is the `Pbind` class, the principal function of which is to combine various streams of information into one *event stream*. Using this facility, it is possible to create 'value patterns' which are mapped to different variables and can be used to perform in creative ways.

More recently, *Conductive* [3] has been designed as a language that offers a higher-level abstraction. In *Conductive*, for example, it is possible to generate *sets* of varying densities based on an initial pattern, and to store them in a table indexed by their level of density, so that they can be retrieved and utilised in compositions[4]. Finally, *TidalCycles* (or *Tidal*, in short) [37, 38] introduced an *embedded Domain Specific Language* (eDSL) for composing patterns as higher order structures with a highly economical syntax. The ideas encapsulated in Tidal and its implementation have greatly informed the effort that is the subject of this thesis, and warrant a more detailed discussion.

1.6.1 TidalCycles

In Tidal, time is conceptualised in a cyclical, rather than linear manner; and the term *arc* is used to describe a temporal range, delimited by specific begin and end time and it is based on rational subdivisions.

Tidal represents each musical event, delimited by a start and stop time that are termed the event *onset* and *offset*, as an *arc*. The association of a value pertaining to two time arcs is termed an *event*. In this case, the first arc is associated with the onset and offset of the event, and a second art relates to its *active* portion. This second arc is utilised in cases where an event consists of multiple pieces – when it is important for each piece to store the original arc that denotes its context. Finally, in Tidal, *patterns* represent functions that associate an arc to a list of events. Patterns can be *queried* with an arc to return a list of all events that are active during a given time. These arcs may have overlapping events, which supports polyphony in music without requiring the introduction of events that deal with multiple values (even though using chords in lieu of atomic events is a possibility).

Conceptually, all Tidal patterns are infinite in length. They can cycle indefinitely, and they can be queried for events at any point in time.

Tidal does not have built-in synthesis capabilities itself; rather, it is designed to be used primarily with the *SuperDirt* sampler [1], and can communicate with other instruments over the *Open Sound Control (OSC)* and *MIDI* protocols.

1.7 Command-line Interface (CLI)

The Command Line Interface (CLI) was the primary means of interaction with most computer systems on computer terminals in the mid-1960s, and continued to be used as the main interaction interface throughout the 1970s and 1980s. The interface is usually implemented with a command line shell, which is a program that accepts commands as text input and converts commands into appropriate operating system functions.

While they capitalise on the CLI, hybrid systems for algorithmic composition that combine multiple approaches led to new possibilities for expression in live-coding [41]. These systems make use of two or more languages and communicate through various bindings like *Open Sound Control* or *WebSocket* [48]. These languages are mostly domain specific hence they can be accessed directly using a CLI and the associated compiler. 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 of different domains together.

Using Tidal often requires an external text editor, which is often not purpose-built for live coding, to be able to interface with a sampler or synthesizer (e.g. SuperCollider). Siren leverages existing live coding practices by providing a textual or "command line" access point, but this is integrated into a graphical ecosystem where patterns can be retrieved from a visible pattern history. It allows user to play along with the timers and intervene to the running patterns hence maintaining the expressivity of live coding.

1.8 Musical User Interfaces

The design of new interfaces, as well as the interaction between forms of music engender a compelling domain of research that links the practices of composition, performance, and improvisation in the context of the computer. Crucially, current real-time systems permit the composer to become the performer of their composition, even directly affecting the micro-structure of sound and affect in a very immediate and direct

way [17]. One important consideration about this creative domain is that the *interface* shapes how the practitioner perceives and interacts with their art. In the formal systems on which this domain is founded (e.g. the implementation of computer programming languages and conventional practices of musical composition) it is precisely the design of *notation* that defines the *affordances*, i.e. what actions and expressions are possible. This notation system is provided by the programmer in the live coding (see Sec.1.4) while in any other conventional music system it is provided by some kind of a "discrete" (abstract but always discrete) grid system.

1.8.1 Musical Trackers

Trackers are a class of sequencer based on a concise text notation, edited using the computer keyboard. The user interface of Siren is heavily influenced by the notion of a musical tracker. 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 organised into a sequential master-list to form a complete composition. 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 [9] in 1987, NoiseTracker [33] in 1989, and Protracker [14] in 1990. Ultimate Soundtracker was the creation of Karsten Obarski, who built it to relieve himself of the *labor* involved in coding computer music by hand, with to tools available to him at the time. Obarski designed a tool that graphically represents the four channels of sound on the Amiga's sound chip like a vertical piano roll. The piano roll metaphor elegantly matched the looping structure common to nearly all music playback subroutines of the SID period [13]. Trackers are concluded as one of the most effective digital instrument in the findings of Nash and Blackwell on *Cognitive Dimensions of Notation*[7] and will be evaluated more profoundly in the next chapter.

Chapter 2

Siren: An Ecosystem for Musical Patterns

This section presents Siren, a musical user interface envisioned as novel *ecosystem for pattern creation and sequencing*.

First, the structure of the pattern creation and sequencing ecosystem that Siren is based on, in terms of its data structures and underlying technologies, is described. This is followed by an introduction to the principles that informed the design and implementation of Siren. Subsequently, a report on the current implementation of Siren user interface, which is intended to function as a software application for both composition and live performance, concludes the section.

2.1 An Ecosystem for Pattern Creation and Sequencing

Siren is an ecosystem for pattern creation, live coding performance, and algorithmic composition. Here, the concept of a "pattern" is borrowed from the Tidal programming language "for encoding musical patterns during improvised live coding performances" [40] (see Sec.1.6.1). In essence, Tidal is a "pattern language" which is "embedded" in the Haskell programming language [22], which offers means to represent the encoding of musical patterns, a "library of pattern generators and combinators" and a scheduling system for dispatching events. Siren is based on a hierarchical structure of data, initially intending to build on the concepts and technology of Tidal. The main idea in Siren 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 [46]. In addition to pattern arrangement, the user interface in

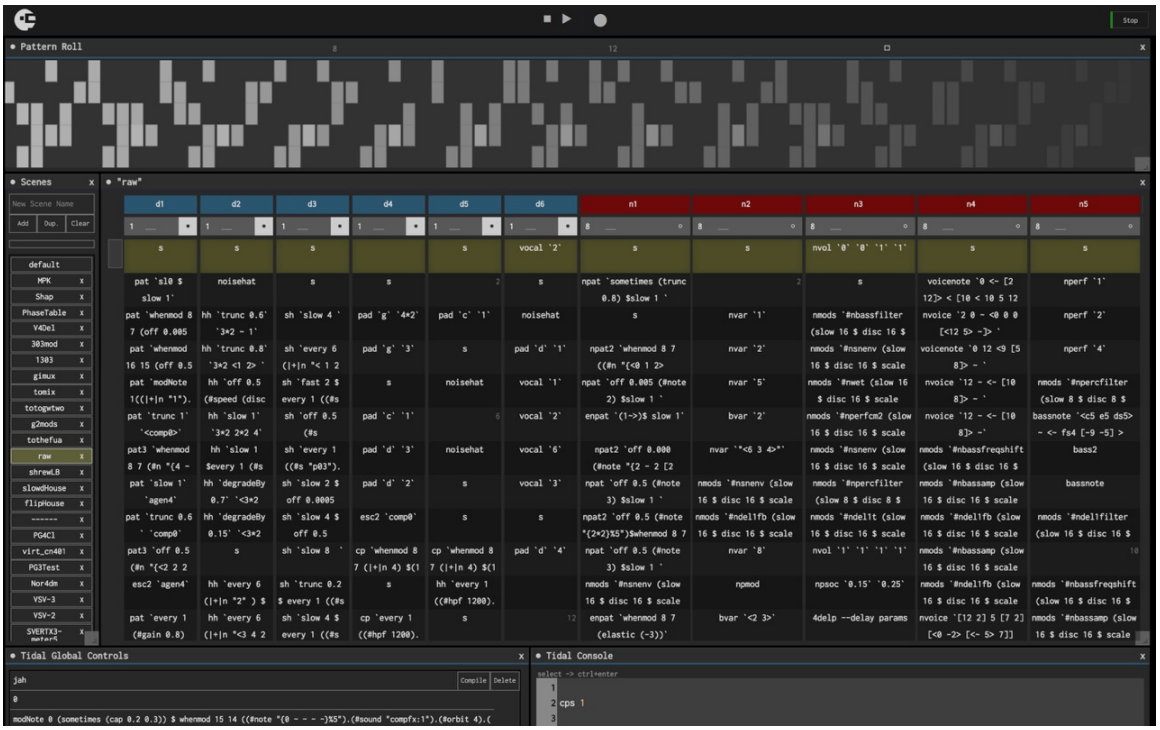

Figure 2.1: Siren, based on a hybrid visual approach that marries live-coding with a tracker-inspired environment, is meant to serve as an ecosystem for musical patterns and pattern-based compositions.

Siren supports the specification of variations and transitions between patterns. Thus, it can be considered an *ecosystem* for patterns and the representation of compositions based on patterns, based on a visual approach to live performance with code, in a familiar, tracker-inspired, grid-based environment (see Fig.2.1).

In a live coding session, a performer usually follows a vertical work-flow to compile the code. Since the top pattern is usually compiled first and the performer keeps adding more variations and channels towards the end of the text editor. This results in an interleaved list of patterns with different channels, including multiple variations of each channel. Tracker in Siren untangles this list by keeping the arrow of time as is and organises the items into multiple channels.

In Siren, patterns denote functions that are stored in a *pattern dictionary*. These pattern functions can be called from cells in the channels, and parameters that affect the behaviour of the pattern functions can be specified within cells for each call.

While the concepts that have inspired Siren are rooted in Tidal, patterns in Siren need not be encoded in the Tidal language. The current version of Siren indeed supports the use of Tidal and SuperCollider but the application of the underlying concept is not limited to these languages, and the software can be extended to use

other means of encoding patterns.

In the tracker grid, per convention, each column represents a channel. Channels in Siren are analogous to "tracks" in contemporary digital audio workstations. In the current implementation, channels are specific to a certain pattern language—while patterns encoded in different languages can be stored in the same pattern dictionary, each channel may only call pattern functions written in a specific language that is determined by the user upon the creation of the channel. In other words, each channel runs an interpreter or compiler for a specific programming language as its back-end. Each cell in a channel can be filled with a single *function call*, the definition of which is found 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 to it.

A function that is defined in the pattern dictionary can be called from any cell of any channel (provided it is written in a language that the channel's interpreter can execute). The execution happens thusly: A timer, or scheduling system, in Siren scans each channel from top to bottom, triggering pattern functions in cells as they are encountered. When a cell is triggered by the timer, Siren performs a look-up in the pattern dictionary. Once the desired pattern is found in the dictionary, 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. For example, in the case of Tidal channels, the Glasgow Haskell Compiler (GHC) [26] is used to compile patterns to be parsed by Tidal, which can then be sent as an OSC [63, 62] message to the SuperCollider software (which runs separately from Siren). For SuperCollider channels, SCLang [54] is used for the communication between the interface and SCsynth.

2.1.1 Hierarchical Composition

The main musical structure in Siren is the concept of the *scene* (see Sec.2.3.2.1), which acts as a top-level container and a framework for a composition. Each scene could be thought of as a grid where each column is a channel and each row denotes time steps. A timer cycles through the rows, from top to bottom, and triggers the content of each cell. Each scene has a *pattern dictionary* (Fig.2.1) for the storage of *pattern function*, their parameters and implementations (Fig.2.6). Instances of these pattern functions (Fig.2.3) can be written into the grid to act as a function call to patterns on trigger. The size of the grid is bound to the number of channels, in

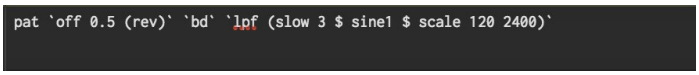

Figure 2.2: Example for an instance of a pattern function call within a channel.

Figure 2.3: Example for a pattern function located in the pattern dictionary.

other words, the number of columns in the grid. A user can create multiple scenes, each of which has unique pattern functions and channels. A *scene* can be used as a sketchbook or it can represent a composition.

2.1.2 Patterns as Functions

The system is designed to treat stored patterns as functions. Patterns are referenced as *function calls* (e.g. Fig.2.2), which is written in one of the cells in the channel, and the *pattern function* (e.g. Fig.2.3), located in the *pattern dictionary*. The pattern function contains the optional parameters and the implementation of the instance. Furthermore, an instance of a function has two components, function name and parameters that are enclosed in grave accents in turn (`).

2.1.3 Features

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

2.1.3.1 Parameters and Modulations

The pattern functions on Siren can contain any number of *literal* or *random* parameters.

A *literal* parameter is defined as any sub-string enclosed in grave accents (except for `‘t’`, which is reserved for the *temporal* parameter (see Sec.2.1.3.2). They can be used in multiple places in the same instance, resulting in the ability to create complex relationships and modulations, especially when used with mathematical expressions (see Sec.2.1.3.3). These parameters do not have a specific type, and any kind of input that reconstructs a syntactically correct pattern is accepted.

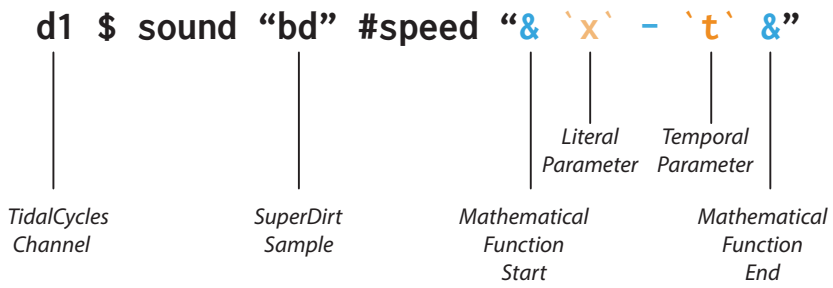

Figure 2.4: The syntax used in Siren

Siren also introduces *random* parameters, which can be constrained within given boundaries (e.g. `|0,3|`) in a cell. This adds an extra element of randomness to the compositions. This aspect of Siren is 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). In other words, bouundaries of randomness in the speed parameter can be set by another bounded random parameter, dispatched from the cell.

```
d1 $ sound "cp" #speed (scale 0.5 ‘x‘ $ rand)
```

In the example above, lower boundary of randomness is clamped with 0.5 while the upper bound can be assigned to different values within the cell. Alternatively, a literal parameter can be used to modulate the functions themselves. In the example below, ‘y’ parameter can be initiated with different functions like `slow` which would slow down the pattern 4 times or `chop` which would slice the sample into 4 pieces.

```
d1 $ ‘y‘ 4 $ sound "cp"
```

The most efficient usage of *parameters* in Siren is yet to be explored. Nonetheless, it’s an effective solution to the shortcomings of Tidal.

2.1.3.2 Polyrhythmic Timers and Temporal Parameter

Polyrhythmic events are possible through the implementation of dynamic channels. Channels can have different numbers of steps, and by incrementing the number of steps, events can be quickly created or extended. The tracker’s timer is represented with a temporal parameter (`‘t‘`), which provides the current time to *pattern functions* at the moment of execution (see Fig.2.3).

```
d1 $ fast ‘t‘ $ sound "cp"
```

In this case, the speed of the pattern would differ depending on the step it was dispatched from. The temporal parameter can also be used together with a *literal parameter*. Modifying the previous example:

```
d1 $ ‘x‘ ‘t‘ $ sound "cp" #speed "& 4 % ‘t‘&"
```

The temporal parameter would supply parametric value to the function specified by ‘x‘ and the pattern would get pitched up to 4 octaves depending on the dispatched cell.

2.1.3.3 Mathematical Expressions

The system allows use of a wide range of mathematical expressions that enhance the computational and algorithmic aspect of pattern creation. These expressions can be employed in any part of the implementation of pattern functions, and upon successful evaluation, the expression is replaced with the final value. These expressions are notated by surrounding the expression with ampersands (`&`). Integration with parameters, especially the constantly incremented temporal parameter (i.e. `‘t‘`), opens up new relationships for certain parts of the pattern, such as:

```
slow $ &log(‘t‘*‘t‘,2)& $ sound "synth" # sustain "&‘t‘*‘t‘&"
```

Here, the duration of the triggers and speed of the pattern are in a relation with channel's `time` value squared– the amount of the traversed cells in the channel, squared (see Sec.2.1.3.2). This implies that the pattern would evolve as the composition progresses. The expressions support numerical spaces, symbolic calculations, trigonometry, vector and matrix arithmetic [27].

2.1.3.4 Transitions

In Tidal, the functionality to specify transitions between patterns is provided. However, the transitions must be specified per pattern. Siren provides the feature to specify a Tidal transition function for each channel only once, and have the transition function affect all of the patterns in the channel.

```
t1 (xfade) $ sound "bd cp"
```

In the example above, each compiled pattern would slowly fade in while the previous one fades out. Alternatively, a pattern can be used as a single shot by utilising the transition function named `mortal` which degrades the new pattern over the given

time. This allows triggering single shots and short patterns which can be used as filler or transitive elements which is commonly used in the conventional trackers (see Sec.1.8.1).

2.1.3.5 Global Modifiers

Another feature of Siren is that there are *global transformations* and *parameters* which act on all of the channels, by either prepending transformation functions or appending parameters to the patterns. Some examples of parameters, `#speed -1` reverses current playback on all channels, `#coarse 2` halves the sampling rate of the selected channels, and so on. The global modifiers can be stored and recalled from the associated channel within the grid interface.

2.1.3.6 Conditional Triggers

Every pattern and trigger returns an acknowledgement upon successful its compilation. This feedback mechanism allows interconnected and conditional events possible in Siren. By parsing the the response messages from SuperCollider, it is possible to keep track of the past events and conditionally trigger others.

```
d1 $ note "0 1 [12 7]" #s "synth"
    #trigLookup "7" #trigEvery "5"
    #trigSound (pure "d2 $s "\"cp bd\"")
```

The parameter specified in `#trigLookup` is the active parameter which increments the counter for the conditional trigger. The required trigger number is specified by `#trigEvery` parameter and the conditional pattern, `#trigSound`, is specified with escaping quotations. In the example above, the 'd2' pattern would be compiled after the fifth trigger of note '7'. Although, the syntax is a bit verbose, it offers an unique way to approach pattern creation.

2.2 User Interface Design Principles

A number of design principles have guided the development of Siren. Its user interface has been inspired by the previous works exploring the psychology of interacting with notations in the field of programming. These works break different factors of the software designer's user experience into cognitive dimensions that help to paint a broad picture of the user experience involved with editing code and crafting software systems.

The first strand of ideas to inform the effort is the *Cognitive Dimensions of Notations* framework, originally introduced by Green for the purposes of investigating the psychology of programming languages [18], and subsequently adapted towards music notation systems and musical user interfaces [7, 6, 47, 45]. The second is the *Design Heuristics for Virtuosity* proposed by Nash and Blackwell [10], which builds on the cognitive dimensions framework and focuses on attributes that support virtuosity in musical creation interfaces.

2.2.1 Cognitive Dimensions of Notations

In any attempt to interact with a musical system, there will be many inescapable limitations. The interaction is always mediated through an abstract layer of notation, e.g. a stave, sequencer or waveform metaphor. The process of *sketching* illustrates how experimenting with notations can be used to support creativity. This encourages greater optimism about the opportunities afforded by notation-mediated music interaction (see [7]). In the words of Nash and Blackwell [47]:

> Notations, and the interfaces used to edit them, may provide a description of end product (be informative), define an exact specification of it (be significant), have editing actions offer rapid feedback (be responsive), or be inseparably and continuously coupled to the product itself (be live). This theory demonstrates how non-realtime, notation-mediated interaction can support focused, immersive, energetic, and intrinsically rewarding musical experiences, and to what extent they are supported in the interfaces of music production software. Users are shown to maintain liveness through a rapid, iterative edit cycle that integrates audio and visual feedback.

Based on the above, a number of *cognitive dimensions* are summarised below, based on the research in cognitive science, but shaped to serve as a practical analysis and guiding tool for interaction designers, researchers, and programming language architects. "Each dimension is intended to describe a distinct factor related to the usability of a particular notation. The goal is for each to relate to properties such as *granularity* (considered on a continuous scale from high to low), *orthogonality* (independence from other dimensions), *polarity* (not necessarily in terms of the 'good' and 'bad,' but characterising 'desirability' in a continuous manner, for a given context), and *applicability* (in terms of a broad relevance to any kind of notation)" [45]. From Green and Blackwell [19], these dimensions, along with their definitions, are listed in Table2.1.

Dimension	Definition
Abstraction	Types and availability of abstraction mechanisms
Hidden dependencies	Important links between entities are not visible
Premature commitment	Constraints on the order of doing things
Secondary notation	Extra information in means other than formal syntax
Viscosity	Resistance to change
Visibility	Ability to view components easily
Closeness	Mapping closeness of representation to domain
Consistency	Similar semantics are expressed in similar syntactic forms
Diffuseness	Verbosity of language
Error-proneness	Notation invites mistakes
Hard mental operations	High demand on cognitive resources
Progressive evaluation	Work-to-date can be checked at any time
Provisionality	Degree of commitment to actions or marks
Role-expressiveness	The purpose of a component is readily inferred

Table 2.1: The Cognitive Dimensions from [19].

Building on these dimensions, Nash formulates a set of questions to guide user interface designs in terms of more concrete design considerations [45]:

1. "How easy is it to view and find elements or parts of the music during editing?"
2. "How easy is it to compare elements within the music?"
3. "How explicit are the relationships between related elements in the notation?"
4. "When writing music, are there difficult things to work out in your head?"
5. "How easy is it to stop and check your progress during editing?"
6. "How concise is the notation? What is the balance between detail and overview?"
7. "Is it possible to sketch things out and play with ideas without being too precise about the exact result?"
8. "How easy is it to make informal notes to capture ideas outside the formal rules of the notation?"
9. "Where aspects of the notation mean similar things, is the similarity clear in the way they appear?"
10. "Is it easy to go back and make changes to the music?"

11. “Is it easy to see what each part is for, in the overall format of the notation?”

12. “Do edits have to be performed in a prescribed order, requiring you to plan or think ahead?”

13. “How easy is it to make annoying mistakes?”

14. “Does the notation match how you describe the music yourself?”

15. “How can the notation be customised, adapted, or used beyond its intended use?”

16. “How easy is it to master the notation? Where is the respective threshold for novices and ceiling for experts?”

These abstract cognitive dimensions and higher-level guiding questions have been one component of the principles on which the design of the interface is based. It would not be tractable to formulate one-to-one mappings between each feature or design element in Siren and a specific cognitive dimension or guiding question, however, below I can offer some comments on how exactly these concepts manifest in the final design. The implementation, final interface design, and how various features relate to the considerations enumerated above will be the examined in detail in the the next section.

In Siren, the approach of the “pattern dictionary”(Fig.2.6) plays a critical role in editing musical compositions. In conventional step sequencers, step editing needs to be done by directly accessing its dedicated *menu* or similarly referenced location within the system. As opposed to that, in Siren there are no constraints as to where the step-wise edit options needs be performed. This makes it easy to immediately find musical elements and parts, which are already laid out in full view.

Siren strives to provide multiple ways for accomplishing the same musical result using different structures and utilising different pattern languages (some are more accurate and more “loyal” than other with respect to an initial idea). *Global modifiers* (see Sec.2.3.2.4) allows direct modifications to the running patterns which could be seen as a tie with the notions of diffuseness and role expressiveness (Table2.1).

The system allows musical ideas to be sketched and re-used within its hierarchical structure (see Sec.2.1.1), and the user interface provides methods for this without adding further distractions to realising musical ideas. The modularity of musical expressions in Siren is an innate feature of the user interface paradigm and layout system.

Siren leverages different abstractions in order to make good use of the visual space. The essence of the notation is initially determined in the pattern language, and its progression corresponds with a secondary timer structure. In terms of progression, these abstractions provides a great amount of freedom, as it's possible to apply modulations to the core parts of the patterns.

Sketching is one of the salient aspects of live coding practice – the code written for a live performance could end up as a sketch after the performance, if not deleted, and serve as the basis for a later composition. When beginning a composition, there is no idea, and hence, no exactness. Findings in the initial experimentation stage lay out the fundamental ideas, which are subsequently implemented in a more exacting form in order to create a composition. In Siren, every scene can be utilised as a sketchbook to incubate compositional ideas. As such, an initial sketch can be created very quickly by either utilising the pattern language (on the console) and using a few channels. From this view, the compositional environment in Siren enables low-viscosity workflows and secondary notations.

Global modifiers (Fig.2.7) in Siren afford global modulations to the all active patterns. Using this feature, implementing any informal decision which would affect the selected patterns is trivial. As such, this enables modulations to parameter values to be implemented using a selection of abstractions, without much premature commitment, while enabling provisionality.

Since Siren is, in its essence, *a layer of abstraction* on text-based programming environments, it is often very easy to go back and make changes to any scene or pattern within seconds which would affect the musical output.

In terms of the notations utilised for each pattern, it is very easy to differentiate the relationship between other patterns by their names and channel placements (one possible limitation is that, within a channel, it may be difficult to identify two instances of the same pattern that uses different parameters).

Edits can be performed in any order or any part of the composition can be theoretically created without listening to it. This ability 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, some 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 utilised to introduce a break or a verse. In a way, it is designed to allow mistakes which may end up fueling inspiration.

Some limitations of the implementation include the absence of micro-tonality and the lack of extensive customizability for the notation. In Siren, currently, the only form of notation that can be used to express fine-grained ideas is by writing code. The upside to this is that code can allow for highly sophisticated creative possibilities.

2.2.2 Design Heuristics for Virtuosity

The design of Siren is further informed by the Nash and Blackwell's *Design Heuristics for Virtuosity* [10], which has the "cognitive dimensions" framework at its foundation, and focuses on foregrounding factors that entail "virtuosity" in the user interfaces for live musical performance. These heuristics are:

1. Support learning, memorisation and prediction ("recall rather than recognition")
2. Support rapid feedback cycles and responsiveness
3. Minimize domain abstractions and metaphors
4. Support consistent output and focused, modeless input
5. Support informal interaction and secondary notation

These heuristics have been chosen due to the notion that rather than more conventional approaches to usability and user experience design in human-computer interaction, user interface designs for supporting musical creativity are better informed by considering issues such as *flow*, *liveness*, and *virtuosity* that are of importance in musical contexts [46, 47, 10].

In Siren, specifically, the principle of "recall rather than recognition (1)" and support for "rapid feedback cycles and responsiveness (2)" have been considered as guiding notions for the design of the user interface, its underlying concepts, and its implementation. Moreover, "support for informal interaction and secondary notation (5)" is allowed through a fusion of *textual* and *visual* approaches to programming and composition. The fusion of these approaches is also visible in consideration of the *dimensionality* of representations in terms of both their visual manifestation and the mental models they encourage.

Textual programming can be considered one-dimensional because the relationships are encoded in text, and thus, are represented based on adjacency. In visual programming (for example, in *Max/MSP* [16] or *PD* [50]), putting objects on top of each other

is possible and the program still runs. However, since the encoding is compiled into machine instructions regardless of its visual representation or its user interface, this dimensionality only pertains to the experience of the user, not to the workings of the process itself [42]. In *textual programming* however, these visual manifestations also represent the workings of the compiler. In 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. Furthermore, in Siren, the notion of supporting "consistent output and focused, modeless input (4)" is addressed through the use of a single view that exposes almost all of the elements of a composition.

While these four heuristics (1,2,4,5) are well-supported in Siren, a concession that has to be made is that in terms of "minimising domain abstractions and metaphors (3)", the design of Siren falls short in that it relies on an integration of two separate domains of musical practice: live-coding and tracker-based composition. However, still, Siren observes this heuristic by striving to follow already established abstractions and metaphors. Therefore, the design ensures that the users who are readily familiar with the live-coding and tracker environments can intuitively start using the main concepts of the system without intricate knowledge about the interface.

2.3 Siren: Implementation and Usage

Siren is a JavaScript-based application. The back-end, which interfaces with GHC[1] and SuperCollider, is built using *Node.js* [15]. *React.js* library [23] was chosen to build the user interface, due to its stability and active user community.

Siren makes use of the TidalCycles (Tidal) domain-specific language which runs embedded in Haskell. The Tidal language is designed for artists to generate and manipulate "audible or visual patterns" by writing code, and accommodates both compositional and live performance workflows. For writing musical patterns in Siren, Tidal was a convenient choice for the reasons that it treats compilation as an evaluation of mathematical functions and avoids changing-state and mutable data. Therefore it is highly practical for pattern programming.

In the initial version, internal states of the software was managed with "Redux" library but re-factored to work with "MobX" in the later version to utilize its bi-directional state management. The bi-directionality allowed Siren to communicate its internal states with the interface more efficiently. As for the pattern highlighting,

[1] Glasgow Haskell Compiler

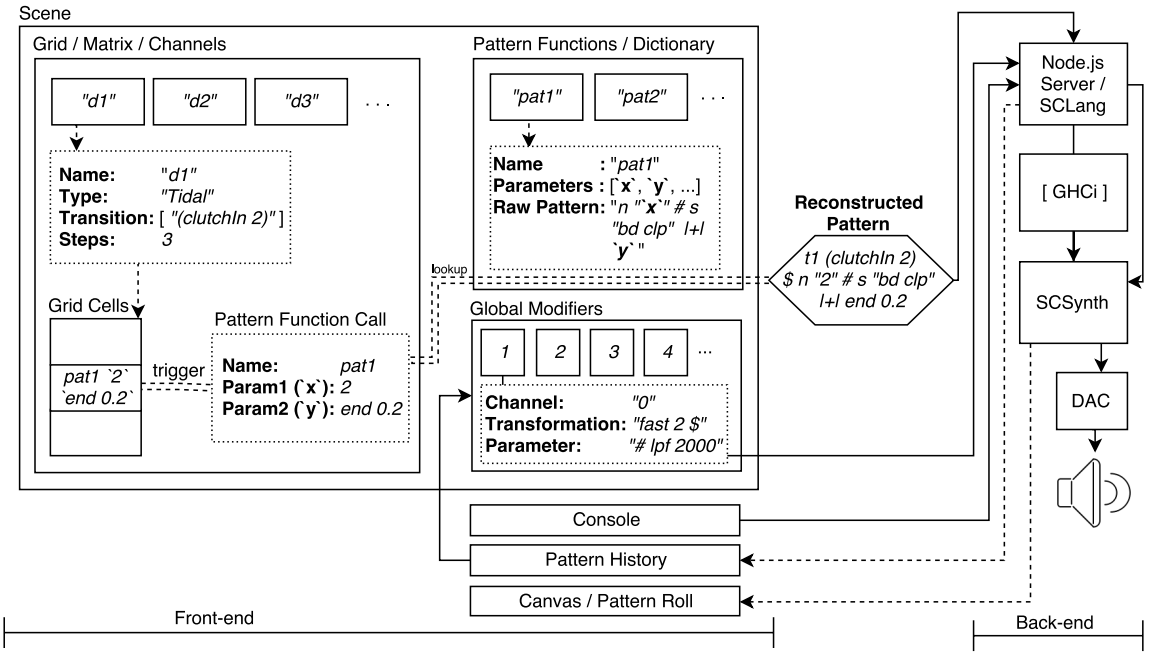

Figure 2.5: The block diagram for Siren – The "front-end" explains the hierarchical structure of Siren, and the communication between different components are shown with arrows. Examples and explanations of the most prominent items are presented with thin dashed boxes. Optional fields are shown with square brackets ([ ]).

Siren utilizes the open-source library CodeMirror, which affords for various features such as syntax highlighting and customised themes.

2.3.1 System Structure

The 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[2] 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.

2.3.2 Modules

In Siren, it's possible for a user to save four customs layouts with different modules. For example, one layout can maximise the channels to take up the full screen, allowing the user to focus on sequencing, or the console could be made to take up the full

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

screen for a user experience similar to a text editor, favoured by programmers. These layouts can be saved within the right-click menu. Navigating between the layouts is also possible using convenient key-bindings.

2.3.2.1 Scenes

Scenes are the essential component of Siren. A scene serves as a container mechanism for patterns and channels. A scene can be added, duplicated or deleted.

2.3.2.2 Channels

Channels can be added using the context menu and consists of “type”, “name”, “step” and an optional ‘transition‘ parameters for initiation. Once a channel is added to the sequencer, the parameters and layout can be adjusted dynamically. Patterns can be looked up from the dictionary with their names and parameters. When a cell in the channel is active, it triggers the pattern with appropriate name and applies parameters in an ordered fashion (see Sec.2.1.3.1). The cells in the channels serve as a canvas for pattern names and pattern parameters. Pressing Enter on the cells selects the cell. Once in selection mode, you can navigate the cells with arrow keys. The selected cell can be compiled with Alt + Enter. Multiple cells can be selected using the Shift + arrow keys which then can be copied and pasted to other parts of the grid.

2.3.2.3 Patterns

Tidal patterns are stored in the ‘dictionary‘. This dictionary is unique for each scene and interacts with the sequencer in terms of parameters and calls.

The syntax to be used for encoding patterns in each entry in the pattern dictionary is determined by the channel definition, which determines the language in which the pattern will be written. One additional feature that is specific to Siren is that parameters can be provided instead of constants (see Sec.2.1.3.1), and specifically when Tidal is used as the pattern language, the channel names must be omitted. Different types of parameters are accepted within the patterns as explained in Section2.1.3.1.

2.3.2.4 Global Controls

The Global Modifiers module allows global control over the running patterns and comprises three functional sections: an *execute* section where the main functionality of the module is controlled and *the memory banks* below which allow saved functions and

Figure 2.6: The pattern dictionary in Siren.

parameters to be recalled. In the *Execute* section there are two sections dedicated to appending and prepending to the running patterns. '*ctrl+enter*' applies the *modifiers* to the patterns. These sections can be saved and recalled by creating presets. Pressing '*save*' button saves the active modifiers. These modifiers are applied to the patterns shown in the pattern history section (i.e. active patterns). Channels that you want to modify can also be specified using the 'channel' section in the sub-menu. Writing the channel names will make the modifiers affect the specified channels, `'0'` is a special case and means that modifiers will be applied to all channels in the scene. One caveat regarding this module is that the design lacks visual clues that would better communicate its functionally, which will be implemented in the future.

2.3.2.5 Playback

Siren allows live recording of patterns to be re-created as a scene after the session. If the record is toggled, each compiled pattern will be time-stamped and written to the disk. After recording the session, this module allows creating a scene from the time-stamped patterns. Each pattern gets decomposed by the selected strings (n, note, and sound), parametrized and placed in the channels.

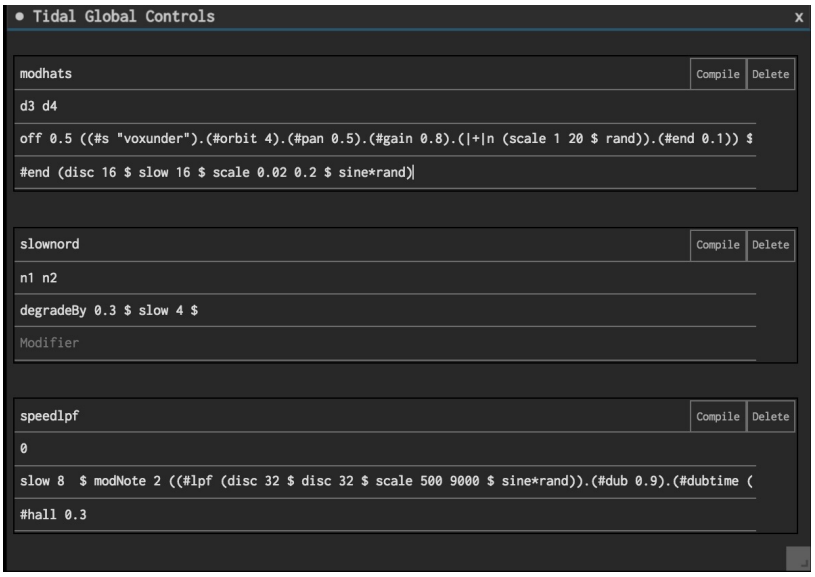

Figure 2.7: The Global Modifiers module.

2.3.2.6 Pattern Roll

It's possible for algorithms to create patterns where it could be too laborious to create by hand. However, while these patterns can be interesting, it sometimes may be hard to decode the relations between different elements. This is to say that the visual aid is occasionally helpful to decode complex patterns.

The Pattern roll (Fig.2.8), is a visualiser for the pattern triggers. This module is dedicated to listening incoming OSC messages from SuperCollider, and it serves as a visual tool to understand relationships between individual triggers. The default sequence length is 8 seconds and each second is quantized into 12 bins. However, both parameters can be edited using the dedicated fields on the interface.

It records the running patterns by listening for the playback messages from SuperCollider OSC function. Upon parsing the messages, note and/or sample numbers are placed in the data structure in which the vertical dimension lists unique samples or notes, and the horizontal dimension denotes the time. In the future, this module will support editing the triggers as well visualise the future events generated by the compilation of code.

2.3.2.7 Pattern History

This module stores successfully compiled patterns and keeps track of the running sequences. In the back-end, this module communicates with the *Global Modifiers* (see Sec.2.3.2.4) module to affect the running patterns.

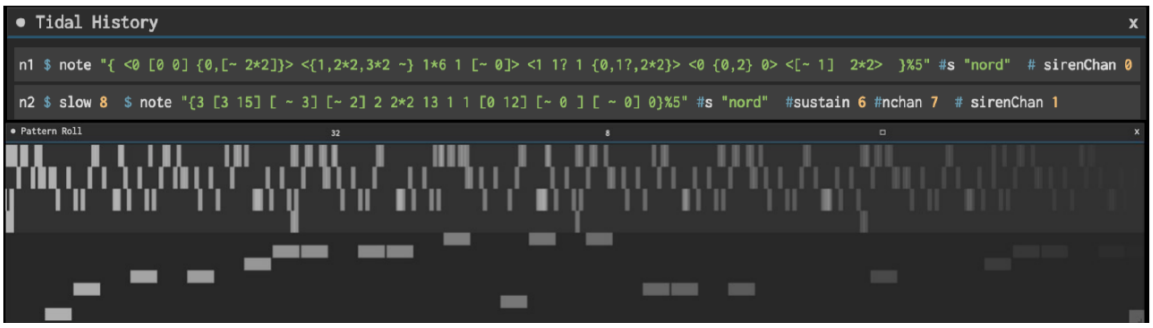

Figure 2.8: The Pattern Roll with two running patterns.

2.3.2.8 Controller

This module is dedicated to communication with *Nord Modular G2*. It requests controller snapshots by parsing the last compiled nvar, which is a variation change message, coming from the cells and maps the returned parameter to the knobs located in the module. These knobs are then used to control parameters in the modular patch by sending MIDI CC messages. Similar to the Global Modifiers (see Sec.2.3.2.4), this module also supports saving, and loading presets.

Figure 2.9: The Pattern History module.

2.3.2.9 Console

This module serves as a *CLI* (Command-Line Interface)(see Sec.1.7) to GHC and SuperCollider. The compiled patterns from this module can be monitored in the *Pattern History* module if appended with 'sirenChan' parameter (see Sec.2.3.2.7)

2.3.2.10 Tidal Log

This module serves as a debug console for GHCi. It plays a crucial role in debugging the complex patterns by displaying the error messages in the user interface.

2.3.2.11 Config Paths

In this module, its possible to set various settings of Siren such as start-up configurations for SuperCollider and Haskell. While it is possible to set these paths within

the file located in the source folder, this module was provided to ease the end user's installation process, which often is one of the significant obstacles on getting started to live coding languages.

2.4 Future Works

There are a number of features planned to be added on top of the current structure of Siren. To sum up those features, the software could benefit from several interface level additions like sliders and curves for different parameters. Especially automation curves would allow creating temporal tasks that would change parameters at regular intervals, independent from the control of the performer [32]. Furthermore, the *Playback* module mentioned in the previous section (see Sec.2.3.2.5) would benefit from a more efficient search algorithm for parsing the time-stamped patterns in order to generate scenes.

In the other hand, I suspect that the way forward to develop a competitive hybrid software is through coding it in a native language to be able to fully utilise computer's power. More robust environments such as QT[3] or JUCE[4] would be more suitable for Siren as *Javascript* runs on a single thread, at least for now. It is also possible to develop an integrated pattern language and a sound engine which is tailored to be used with this interface. It would be much more intuitive for non-programmers as the initial setup requirements would be reduced. A helper installation tool would allow for an easier end-user installation process by removing the path configuration process.

In theory, it is also possible to support any number of pattern languages within the Siren interface, such as FoxDot[5]. Concretely, a more interactive user interface with various zoom functions could also improve user experience.

[3] qt.io
[4] juce.com
[5] foxdot.org

Chapter 3

Reflections

In this chapter, I will analyse the methods I used in musical creation whilst composing, as well as examining the underlying pattern structures and features of Siren that influenced the musical output. The techniques and patterns discussed in the next section can be heard in the audio files provided under the folder "Raw-cuts".

In both the creation of Siren and the creation of music using Siren, I worked towards creating structures in novel experimental forms. As I was building Siren, I aimed to focus on automation and abstractions (see Sec.1.3) and there are many different practices for approaching abstractions when creating a piece or in a live performance. How much of the software or the patterns should be represented in a higher level of abstraction? How precise should the editing be?

Following a spirit of experimentation centred around answering these questions, part science and part art, in which I hypothesise on chains of digital signal processing and patterns that are possible to create with my set of tools. Then, I test my hypotheses by programming the patterns to see if they work, and where they lead creatively. My efforts in and around software design have led me to try many different methods, both technically and musically (see Sec.2.2). My argument is that the method one chooses in his/her creative process shapes the musical output, as it propagates in a path which is shaped partly by imagination and by a different way of hearing, creating and experiencing the sounds of the world. Moreover, my composition practice is self-taught over several years of working with different sequencers and systems, fiddling with the menus and notations, analysing and understanding their relations. The diagram below inspired by the Bricolage Programming in the Creative Arts [39] further sums up my workflow with Siren.

In a more critical perspective, this rapid prototyping and feedback cycle makes it harder for me to finish compositions in a single scene. At the same time, it is

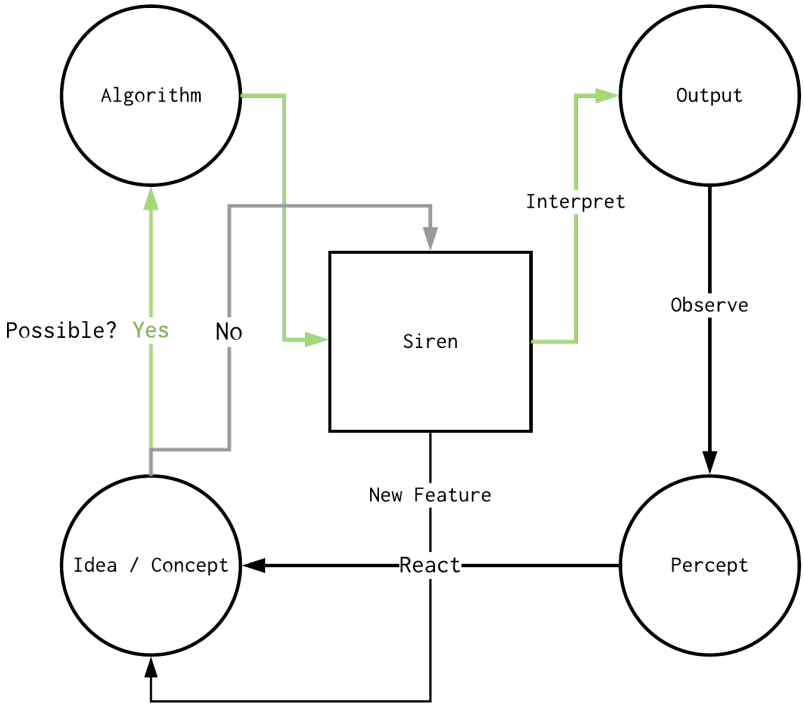

Figure 3.1: The feedback mechanism centred around Siren

undeniable that the rate of improvement in the software for the time being (feature-wise) has a direct relation with the sonic depth of the musical output.

3.1 Musical Flow

In this section, I will be discussing various aspects of the musical patterns and my approach to them.

In Siren, the main focus is on "debugging on-the-spot", in reference to just-in-time (or edit-and-continue) debugging where a running program can be stopped and edited, and can then continue execution without needing to restart[47]. The frequency with which I move between editing the pattern, listening to the output and filling up the steps with variations of patterns has become a well-learnt, reflexive motor sequence, almost an instinctive response to the creation of new material. Playback, while manually triggered, thus becomes tightly coupled with editing, enables a form of *liveness* in my process. This iterative technique allows me to quickly *sketch* and experiment with different ideas. I approached patterns with an aim to use them with different transformers and seeing how flexible I could make them in terms of

being able to reuse them in different contexts. I concentrated on melody, harmony counterpoint with the use of atonal percussive elements. It's usually a conversation with the system before settling down to a structure.

1. Program the initial patterns in the console module.
2. Define the number of channels and pattern functions in the dictionary to be used in the composition.
3. Compose patterns within the channels by modulating their parameters. At each step, discern the quality of the compiled pattern.
4. Create several sections within a scene or inherit another scene from the current one.
5. Improvise with the global modifiers while recording the scene.

Syncopation is a key aspect in the rhythmic patterns employed in my compositions. In short, It stands for displacement of regular triggers. Syncopation in Tidal patterns can be achieved using subdivisions. Variations of these subdivisions create a dynamism on the rhythmic perception of patterns. However, these variations may still sound static without any *unanticipated events*. This brings up the question of the extent to which the repetition in music must be consciously perceivable for it to provide the desired experience. Although, the perception of repetition mostly relies on the rhythmic structure of the patterns, it also feeds from the sound sources and design. In that sense, this perception becomes just another parameter to modulate by step-wise parameterization feature of Siren. I've used algorithmic transformations extensively to redefine the rhythmic structure of the patterns to achieve various musical forms and variations. The most notable modulations, in this case, are changes of the rhythmic density and sound sources. Being able to modulate a programming language unquestionably expanded my creative process by lifting the limitations on modulations and progressions. One of the most interesting examples is to be able to modulate the transformers.

```
n1 $ ‘x‘ 0.5 $ note "{0 ~ 1 2?}%3" #s "nord"
```

In the pattern above, the `x` parameter can be instantiated with different transformers such as 'degradeBy' or 'slow', allowing step-wise modulations to create phrases and variations. It is also one of the features that shifts the approach to Tidal as the steps can be used instead of 'every' function to achieve a similar function ordering.

Moreover, the curly braces ({}) seen in the pattern above tend to produce more interesting behaviour for the rhythmic patterns, but by syntax, it requires a hard-coded number after the symbol. Note that, the *modulo* symbol, in this case, represents the number of *subdivisions* in a cycle to apply the values in the pattern, not to be confused with the mathematical modulo operator. In Tidal, there isn't a way to change the subdivision, aside from using *slow* or *fast* to alter the pattern speed while changing the number. Siren enables the change of subdivisions by combining literal parameters (see Sec.2.1.3.1) with mathematical expression (see Sec.2.1.3.3). For instance, the rhythmic variations in the recording namely "RFVb-2020" were achieved by using this feature. More specifically, the rhythmic alterations can be distinctly heard between 02.00 and 03.00 minutes of the recording. The percussion and the bass in this track hold a nice polyrhythmic relation with bass running on 4/4 and percussion's meter getting modulated between 4/5, 6/8 and 7/9 while it's mainly running on 4/5 time signature.

The pattern shown above can be customised by adding two more parameters to it in order to achieve this effect.

```
n1 $ fast & 'y' / 'm' & $ `x` $ note "{0 ~ 1 2 }%'m'" #s "nord"
```

where the value of 'y' can be constant value that determines the overall speed of the pattern and 'm' determines the subdivisions and adjusts the speed of the pattern accordingly. Moreover, each dispatched event from the code step sequencer can contain different parameters to construct the rhythmic structure, going back and forth between different signatures gives the musical flow an interesting angle to plug into as a listener.

The flow described in this section feeds from the philosophical thoughts on *linear* and *cyclic* time relations to the *musical patterns* explored in the first chapter (see Chapter.1). Both *metric* changes and unanticipated transitions of patterns establish a balance between the linear and cyclic (see Sec.1.5). *Polyrhythms* and otherwise ambiguous rhythms can thus be seen as presenting to the listener a bistable percept [49] that affords rhythmic tension and embodied engagement. Furthermore, the N cycle is a powerful constraint on metric structure and complexity of Tidal patterns. By default, the number of strings (N) specified in the rhythmic structure directly affects the speed of the patterns. Which means that it's possible to represent the same pattern in more than one form. This constraint can be further explored in Siren by parametrization of the rhythmic structure. Modifying the pattern above, with

an additional parameter ‘k’, it’s feasible to utilise the same pattern in the different channels for representing various elements.

```
n1 $ fast & ’y’ / ’m’ & $ ‘x‘ $ note "{‘k‘ }%’m’" #s "nord"
```

Another method that I’ve found effective is sound source modulation. Combining this with Siren’s features can result in complex textural development, but it also may cause a distraction to the performer as the number of available samples increases. Also, the process of making percussive kits from large sets of samples is overly time-consuming and circumlocutory that there is research on automating this process by introducing audio descriptors as discussed by Alex Harker in his writings [20]. I found myself suffering from that very often and wrote a script to index the samples with appropriate convention and create drum kits from that. As an example, the folder which contained the “snares”, tagged with an index of “1”, “claps” folder indexed with “2” and so on. The generated folders were named and indexed as *gen(i)*.

```
d1 $ n "{0 ~ 1 , 2}%4" #s "gen‘x‘"
```

With this folder structure, it becomes possible to change the sound source by applying different integers to the parameter `x` from cells (see Sec.2.1.3.1) to target different sound sources. While this method has been useful for the initial experiments, later on I started focusing on different synthesis methods rather than sample-based percussion.

3.2 Setup and Commentary

In this section, I will briefly explain the patterns and DSP chains I used to create the output included within the submissions.

After I established Siren’s core structure, I felt that I needed to increase my sonic capabilities as Tidal’s approach to sound design is solely based on triggers. I felt limited by this and built a synthesis environment in a Nord Modular G2 [8] aiming to manoeuvre through this limitation. I developed a strict hierarchical paradigm in G2, and am still in the progress of populating the DSP variations within this system. Included in the submission, the “Software/Siren/extras/tidal” folder contains the MIDI mappings used to communicate with this system. The patch structure usually contains a 4-voice drum machine with associated effects, a bass, and a synth voice. All of these sub-patches have eight variation slots and can be activated by sending various MIDI messages. I was loyal to the convention with numbers for the rhythmic patterns,

mentioned in the previous section (see Sec.3.1). In this case, '0' represents the bass-drum, '1' represents snare, '2' bass and finally '3' is mapped to the hi-hats. The bass patch contains a built-in step-sequencer that sends note messages to the patch, triggered by percussion messages. And lastly, the voice patch is as a polyphonic FM synth. A session often starts with tuning the patch using the *Controller* module which contains 64 knobs, mapped to the parameters in the patch. This module perhaps can be seen as the only place within the interface that aims to evoke the feeling *of human presence* [59]. Soon, this module will serve as the foundation for extending the *liveness* and *embodiment* in Siren. Furthermore, this module provides a good oversee of the potential modulation targets as these parameters are also available to modulate from Tidal.

Going back and forth with the routines and techniques mentioned in the previous chapter, I was able to create a sense of liveness which reflects itself in the recordings under the folder "Raw-cuts". To some extent, these recordings are the manifestation of the ideas on *rhythm* and *repetition* discussed in the first chapter (see Sec.1.5.1). The intention behind them was to take the various strategies of repetition and come up with a unique pattern vocabulary that relies on a subtle appearance of rhythmic change over time. A quick glimpse or a few seconds may sound dull but some stretch of time often reveals startling emotional depths.

Even though channels in Siren can be set to stop once they reach the last step, I often found myself *sculpting algorithms* until I degrade them over time. I work from a bank of materials in a variety of combinations. To some degree, most of the output resembles a codified track which is being split up and reconstructed in each session. I agree with Tremblay that improvisation is not a means to generate material to be filtered later on by the composer; it is rather an art of composing in real-time [59]. However, without any post-production, the output lacked the satisfactory energy. As such, I opted for an additional step of mixing and other post-processing.

The recordings explained in the the next three subsections demonstrates the progression of musical output.

3.2.1 1sc34dl

This track is one of the earlier sample-based recordings. My goal was to map cyclical and linear time relations discussed in the first chapter (see Sec.1.5) to interactions of the sounds. The aim was to evoke the feeling of an individual event being pushed further while other prior events are being mashed together in a static yet riveting way. It was generated by utilising several channels with a small number of step,

looping through the events. This recording was primarily an experiment with an ongoing feature at that time, the *global sequencer*, which randomly selects saved global parameters and applies them to the channels with different time intervals. The sudden triggers in the track is an effect of this sequencing silencing the patterns after the specified interval and re-triggering them. Due to the lack of visual feedback of this sequencer, it was removed from the module in a later version. Furthermore, the short recording was refined with dynamic sound manipulation tools and processed until it reached a satisfactory level.

3.2.2 Foilcut

This track was aimed at creating a sense of repetition with the use of atonal percussive sounds, generated in SuperCollider. The idea is for the dense rhythmic triplet pattern to run under a noisy but tonal material. Introduced materials are rhythmically and harmonically in contrast with the main pattern hence functions as a filler in relation to it. The pattern changes in the voice and the bass patches determines the sections in the composition while momentarily changing the overall timbre and degrading the percussive patterns. The relation between meters of the patterns is the main aspect that determines the overall groove of the composition which was achieved by the subdivision modulations explained in the previous section (see Sec.3.1). Noisy hats and FM sounds runs on four quarter-notes as opposed to main pattern's *meter*, which is being modulated between three and five quarter-notes. There are also effects such as *comb* filters and waveshapers being applied to patterns by using the global modifiers (See Fig.2.7).

3.2.3 Gimux

This track is mixed at a friend's studio[1] at Clapton, in London. The main reason for that was to experiment with Siren in a professional studio environment and record the previously stored materials. Patterns in this track are a combination of multiple scenes in Siren, collected over several weeks. The first element is a chord imitating an elastic string until it evolves into a raspy noise with the start of the percussive pattern. The percussive patterns used in the composition can be partially heard in the "Raw-cuts" folder, included in the submission. These patterns were grooved to the multiple G2 performances, each with their variations, triggered and modulated by Tidal through Siren's *scenes*, *globals* and *controller*. After several live coding sessions

[1] https://www.discogs.com/artist/775338-Amir-Shoat

and about a dozen stems were recorded, it was sculpted into its current layered and crowded form. Each of these sessions was enhanced with hardware compressors during the recording phase. This process could be interpreted as the experience of multiple takes within a conventional studio session. Interestingly, most of the transitions were done with timed code executions using Siren's timeline. Effort in the editing mostly went into placing the stems to glue together the composition. The track was then finally rendered onto stereo, showcasing the state of possibilities in pattern creation and sound design with Siren.

Chapter 4

Conclusions

This thesis described efforts to in and around designing a musical user interface envisioned as novel *ecosystem for pattern creation and sequencing*. Throughout this journey, I've encountered various ideas, and by analysing some of them, I was able to make conscious decisions how I would like to utilise programming in music.

Siren has gained significant attention from the live-coding community and changed considerably from being a simple interface for live coding performance towards becoming an open-source music instrument. The system is still in development, co-evolving with my musical creations and with input from a community at large, from which it has received praise and acclaim. I have been invited for the first time to perform at Algorave's "5th birthday online live stream" [24], with Siren. It has been subsequently featured in the TOPLAP website, "the home of live coding" [36]. For the academic community we have published two conference papers at the 2017 International Computer Music Conference (ICMC) titled *Siren: Hierarchical Composition Interface* [25] and at the New Interfaces for Musical Expression 2018 (NIME) conference titled *Siren: Interface for Pattern Languages* [58]. In addition to the conference proceedings, Siren has been featured in a crowd-sourced book on electronic music instruments, *Push Turn Move* [5]. I have performed with Siren in various venues and events, most notable ones are the *London Algorave at the Glove that Fits*, *Sheffield Algorave at DINA* and *the RAW and the Cooked festival*[1]. An album of compositions created primarily with Siren is also in the works. I have also been asked to be in the program committee of the ACM SIGPLAN International Workshop on Functional Art, Music, Modelling and Design 2018 (FARM) and International Conference on Live Coding 2019 (ICLC) through my involvement in the live coding community. On a more critical note, the multiple timers and visualizer in Siren may increase the load

[1] http://therawandthecooked.intersections.io

on the graphics processing unit (GPU). Especially on the inferior GPUs, it is observed to perform poorly when combined with a video capture software.

From the perspective of my personal musical and creative journey, after developing Siren, I can't bring myself to use a traditional DAW for composing and performing anymore, although DAWs retain their place in my workflow for mixing and mastering. The way I work with Siren feels much more efficient and flexible. I get very excited about the idea of opening up programmatic descriptions of musical data because it opens up many opportunities for complexity and variation that are difficult to explore otherwise, especially since this is something I've struggled with ever since I started making music. I believe that computer music is so interesting because it removes many limitations in regards to how one can create and organise sounds. But I also acknowledge that I'm the most productive and creative when I am working within a system with very well defined boundaries and limitations. Artists imposing arbitrary restrictions on themselves to encourage creativity is a common practice in a slew of creative domains. In my experience, there is a middle ground here, where the most productive path to this flavor of creativity is the artists empowering themselves with the knowledge and tools to imagine and create their own systems for artistic experimentation. As such, in Siren, the aim has been to achieve a balance in the design of a musical environment concerning constraints and possibilities of live coding. From my own experience as a composer and performer using the software, I have been able to see that a big part of it has been achieved.

References

[1] Julian Rohrhuber Alex McLean. Superdirt. https://github.com/musikinformatik/SuperDirt, 2015.

[2] HornerImran Amrani. Run the code: is algorave the future of dance music? *The Guardian*, Oct 2017.

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

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

[5] Kim Bjorn. Push turn move. https://www.pushturnmove.com/. Accessed: 2017-12-19.

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

[7] Alan F Blackwell, Thomas RG Green, and Douglas JE Nunn. Cognitive dimensions and musical notation systems. In *Proceedings of International Computer Music Conference, Berlin*, 2000.

[8] Clavia. *Nord Modular G2.* https://www.soundonsound.com/reviews/clavia-nord-modular-g2. Accessed: 2017-03-30.

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

[10] Karen Collins, Bill Kapralos, Holly Tessler, Chris Nash, and Alan F. Blackwell. Flow of creative interaction with digital music notations.

[11] Nick Collins, Alex McLean, Julian Rohrhuber, and Adrian Ward. Live coding in laptop performance. *Organised sound*, 8(3):321–330, 2003.

[12] Edsger Wybe Dijkstra. Notes on structured programming, 1970.

[13] Kevin Driscoll and Joshua Diaz. Endless loop: A brief history of chiptunes. *Transformative Works and Cultures*, 2, 2009.

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

[15] Node.js Foundation. *Node.js*. http://nodejs.org/, 2018. Accessed: 2017-03-30.

[16] francois. A brief history of max. http://freesoftware.ircam.fr/article.php3?id_article=5, 2009. Accessed: 2017-11-27.

[17] Anastasia Georgaki. The grain of xenakistechnological thought in the computer music research of our days. In *Definitive Proceedings of the International Symposium Iannis Xenakis*, 2005.

[18] 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.

[19] Thomas Green and Alan Blackwell. Cognitive dimensions of information artefacts: a tutorial. In *BCS HCI Conference*, volume 98, 1998.

[20] Alexander Harker. Navigating sample-based music: Immediacy and musical control in recent electronic works. 2012.

[21] Andrew Horner and David E Goldberg. Genetic algorithms and computer-assisted music composition. *Urbana*, 51(61801):437–441, 1991.

[22] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A history of haskell: being lazy with class. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 12–1. ACM, 2007.

[23] Facebook Inc. *React*. https://facebook.github.io/react/, 2018. Accessed: 2017-03-30.

[24] Can Ince. Algorave 5th birthday performance. https://www.youtube.com/watch?v=g2dINLvLr1g&t=20. Accessed: 2017-12-19.

[25] Can Ince and Mert Toka. Siren: Hierarchical composition interface. In *Proceedings of International Computer Music Conference, Shanghai*, 2017.

[26] SL Peyton Jones, Cordy Hall, Kevin Hammond, Will Partain, and Philip Wadler. The glasgow haskell compiler: a technical overview. In *Proc. UK Joint Framework for Information Technology (JFIT) Technical Conference*, volume 93, 1993.

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

[28] D Knuth. The art of computer programming 1: Fundamental algorithms 2: Seminumerical algorithms 3: Sorting and searching. *MA: Addison-Wesley*, page 30, 1968.

[29] Charles W Krueger. Software reuse. *ACM Computing Surveys (CSUR)*, 24(2):131–183, 1992.

[30] Henri Lefebvre. *Rhythmanalysis: Space, time and everyday life*. A&C Black, 2004.

[31] Justin London. *Hearing in time: Psychological aspects of musical meter*. Oxford University Press, 2012.

[32] Thor Magnusson. Improvising with the threnoscope: integrating code, hardware, gui, network, and graphic scores. In *Proceedings of the Internation Conference on New Interfaces for Musical Expression*. Goldsmiths University of London, 2014.

[33] mandarin. *NoiseTracker V2.0 by Mahoney & Kaktus*. http://www.pouet.net/prod.php?which=13360, 2018. Accessed: 2017-03-30.

[34] James McCartney. Supercollider: a new real time synthesis language. In *Proc. International Computer Music Conference (ICMC96)*, 1996.

[35] James McCartney. Rethinking the computer music language: Supercollider. *Computer Music Journal*, 26(4):61–68, 2002.

[36] Alex McLean. Toplap. https://toplap.org/. Accessed: 2017-12-19.

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

[38] Alex 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.

[39] Alex Mclean and Geraint Wiggins. Bricolage programming in the creative arts. *22nd Psychology of Programming Interest Group*, 12 2010.

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

[41] Alex McLean and Geraint A Wiggins. Texture: Visual notation for live coding of pattern. In *ICMC*, 2011.

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

[43] John S Mill. A system of logic ratiocinative and inductive london. *Robson, JM (Hg.), Collected Works of John Stuart Mill*, 7, 1843.

[44] George A. Miller. Wordnet: A lexical database for english. *Commun. ACM*, 38(11):39–41, November 1995.

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

[46] Chris Nash and Alan Blackwell. Tracking virtuosity and flow in computer music. In *ICMC*, 2011.

[47] Chris Nash and Alan Blackwell. Liveness and flow in notation use. In *NIME*, 2012.

[48] George Papadopoulos and Geraint Wiggins. Ai methods for algorithmic composition: A survey, a critical view and future prospects. In *AISB Symposium on Musical Creativity*, pages 110–117. Edinburgh, UK, 1999.

[49] Jeff Pressing. Black atlantic rhythm: Its computational and transcultural foundations. *Music Perception: An Interdisciplinary Journal*, 19(3):285–310, 2002.

[50] Miller Puckette et al. Pure data: another integrated computer music environment. *Proceedings of the second intercollege computer music concerts*, pages 37–41, 1996.

[51] Miller Puckette, David Zicarelli, et al. Max/msp. *Cycling*, 74:1990–2006, 1990.

[52] Julian Rohrhuber, Alberto de Campo, and Renate Wieser. Algorithms today notes on language design for just in time programming. In *International Computer Music Conference*, page 291, 2005.

[53] Claude Rostand. Metastasis. *lannis Xenakis: Metastasis, Pithoprakta, Eonta, recording (New York, NY: Vanguard Recoding Society)*, 1967.

[54] Chris Sattinger. Supercolliderjs. https://crucialfelix.github.io/supercolliderjs/.

[55] Mary Simoni. Algorithmic composition: a gentle introduction to music composition using common lisp and common music. *SPO Scholarly Monograph Series*, 2003.

[56] C.J. Sippl and C.P. Sippl. *Computer dictionary and handbook*. H. W. Sams, 1972.

[57] L Spiegel. Distinguishing random, algorithmic, and intelligent music. *Internet: http://retiary. org/ls/writings/alg_comp_ltr_to_cem. html*, 1989.

[58] Mert Toka, Can Ince, and Mehmet Aydın Baytas. Siren: Interface for pattern languages.

[59] Pierre Alexandre Tremblay. Mixing the immiscible: Improvisation within fixed-media composition. In *Proceedings of the Electroacoustic Music Studies Conference Meaning and Meaningfulness in Electroacoustic Music, Stockholm, June 2012.*, 2012.

[60] Raymond Turner and Amnon H Eden. The philosophy of computer science: introduction to the special issue, 2007.

[61] Peter Worth and Susan Stepney. Growing music: musical interpretations of l-systems. In *Workshops on Applications of Evolutionary Computation*, pages 545–550. Springer, 2005.

[62] Matthew Wright. Open sound control: an enabling technology for musical networking. *Organised Sound*, 10(3):193–200, 2005.

[63] Matthew Wright, Adrian Freed, et al. Open soundcontrol: A new protocol for communicating with sound synthesizers. In *ICMC*, 1997.

[64] I. Xenakis. *Formalized Music: Thought and Mathematics in Composition.* Harmonologia series. Pendragon Press, 1992.