

L-Music: an approach to Assisted Music Composition using L-Systems

Ricardo Gil da Silva Durão

**Dissertation to obtain the master's degree in
Computer Engineering, Specialization Area in Graphics Systems and
Multimedia**

Advisor: Doctor João Paulo Jorge Pereira

Porto, October 2021

For Margarida and Rita

For those who find solace and peace of mind in music

For those who care

Abstract

Generative music systems have been researched for an extended period of time. The scientific corpus of this research field is translating, currently, into the world of the everyday musician and composer. With these tools, the creative process of writing music can be augmented or completely replaced by machines.

The work in this document aims to contribute to research in assisted music composition systems. To do so, a review on the state of the art of these fields was performed and we found that a plethora of methodologies and approaches each provide their own interesting results (to name a few, neural networks, statistical models, and formal grammars).

We identified Lindenmayer Systems, or L-Systems, as the most interesting and least explored approach to develop an assisted music composition system prototype, aptly named *L-Music*, due to the ability of producing complex outputs from simple structures. L-Systems were initially proposed as a parallel string rewriting grammar to model algae plant growth. Their applications soon turned graphical (e.g., drawing fractals), and eventually they were applied to music generation.

Given that our prototype is assistive, we also took the user interface and user experience design into its well-deserved consideration. Our implemented interface is straightforward, simple to use with a structured visual hierarchy and flow and enables musicians and composers to select their desired instruments; select L-Systems for generating music or create their own custom ones and edit musical parameters (e.g., scale and octave range) to further control the outcome of *L-Music*, which is musical fragments that a musician or composer can then use in their own works.

Three musical interpretations on L-Systems were implemented: a random interpretation, a scale-based interpretation, and a polyphonic interpretation. All three approaches produced interesting musical ideas, which we found to be potentially usable by musicians and composers in their own creative works.

Although positive results were obtained, the developed prototype has many improvements for future work. Further musical interpretations can be added, as well as increasing the number of possible musical parameters that a user can edit. We also identified the possibility of giving the user control over what musical meaning L-Systems have as an interesting future challenge.

Keywords: algorithmic music, assisted music composition, L-System, user interface design

Resumo

Sistemas de geração de música têm sido alvo de investigação durante períodos alargados de tempo. Recentemente, tem havido esforços em passar o conhecimento adquirido de sistemas de geração de música autónomos e assistidos para as mãos do músico e compositor. Com estas ferramentas, o processo criativo pode ser enaltecido ou completamente substituído por máquinas.

O presente trabalho visa contribuir para a investigação de sistemas de composição musical assistida. Para tal, foi efetuado um estudo do estado da arte destas temáticas, sendo que foram encontradas diversas metodologias que ofereciam resultados interessantes de um ponto de vista técnico e musical.

Os sistemas de Lindenmayer, ou *L-Systems*, foram selecionados como a abordagem mais interessante, e menos explorada, para desenvolver um protótipo de um sistema de composição musical assistido com o nome *L-Music*, devido à sua capacidade de produzirem resultados complexos a partir de estruturas simples. Os *L-Systems*, inicialmente propostos para modelar o crescimento de plantas de algas, são gramáticas formais, cujo processo de reescrita de *strings* acontece de forma paralela. As suas aplicações rapidamente evoluíram para interpretações gráficas (p.e., desenhar fractais), e eventualmente também foram aplicados à geração de música.

Dada a natureza assistida do protótipo desenvolvido, houve uma especial atenção dada ao design da interface e experiência do utilizador. Esta, é concisa e simples, tendo uma hierarquia visual estruturada para oferecer uma orientação coesa ao utilizador. Neste protótipo, os utilizadores podem selecionar instrumentos; selecionar *L-Systems* ou criar os seus próprios, e editar parâmetros musicais (p.e., escala e intervalo de oitavas) de forma a gerarem excertos musicais que possam usar nas suas próprias composições.

Foram implementadas três interpretações musicais de *L-Systems*: uma interpretação aleatória, uma interpretação à base de escalas e uma interpretação polifónica. Todas as interpretações produziram resultados musicais interessantes, e provaram ter potencial para serem utilizadas por músicos e compositores nos seus trabalhos criativos.

Embora tenham sido alcançados resultados positivos, o protótipo desenvolvido apresenta múltiplas melhorias para trabalho futuro. Entre elas estão, por exemplo, a adição de mais interpretações musicais e a adição de mais parâmetros musicais editáveis pelo utilizador. A possibilidade de um utilizador controlar o significado musical de um *L-System* também foi identificada como uma proposta futura relevante.

Palavras-chave: música algorítmica, composição musical assistida, L-System, design da interface do utilizador

Acknowledgements

I would like to start by expressing my gratitude to my advisor, Doctor João Paulo Jorge Pereira, for his unrelenting and unending support throughout the project. His insights were always valuable, and his feedback was always provided promptly and in detail to help me proceed in the best way possible.

I would like to thank my family next, for supporting me when needed. Particularly my dad, for introducing me to Lindenmayer Systems when I was researching for a thesis topic I genuinely found interesting.

A special thanks to João Moreira, Miguel Duarte and João Dionísio, for being readily available to discuss ideas, topics and help me test my work.

Last but not least, I express my deepest thanks to Margarida Guerra and Rita Sousa for being with me for two years and always supporting me and putting up with me when times were low and high. Without your support I would have not been able to finish this journey.

Index

| | | |
|----------|--|-----------|
| 1 | Introduction | 1 |
| 1.1 | Context..... | 1 |
| 1.2 | Algorithmic music composition | 2 |
| 1.2.1 | Autonomous and Assisted Music Composition | 3 |
| 1.2.2 | On the pursuit of computer-generated art | 3 |
| 1.3 | Problem Description | 6 |
| 1.4 | Goals..... | 7 |
| 1.5 | Outline | 7 |
| 2 | State of the art..... | 9 |
| 2.1 | Musical Information Abstraction | 9 |
| 2.2 | Nonadaptive Algorithmic Music Composition | 10 |
| 2.2.1 | Lindenmayer Systems..... | 11 |
| 2.2.2 | Cellular Automata..... | 13 |
| 2.2.3 | Swarm Intelligence..... | 15 |
| 2.3 | Adaptive Algorithmic Music Composition..... | 15 |
| 2.3.1 | Explicit Modeling | 16 |
| 2.3.2 | Implicit Learning | 17 |
| 2.3.3 | Evolutionary Approaches..... | 19 |
| 2.4 | Notable Examples | 20 |
| 2.4.1 | Melomics..... | 20 |
| 2.4.2 | Continuator..... | 21 |
| 2.4.3 | Manousakis' Musical L-Systems | 21 |
| 2.4.4 | DeepBach | 22 |
| 2.4.5 | FlowMachines | 22 |
| 2.4.6 | Dadabots..... | 22 |
| 2.5 | Summary | 23 |
| 3 | Business Value Analysis | 25 |
| 3.1 | Assisted Music Composition Systems as a product..... | 25 |
| 3.2 | Innovation Process | 26 |
| 3.2.1 | Opportunity Identification & Analysis | 26 |
| 3.2.2 | Value Creation | 27 |
| 3.2.3 | Value Proposition..... | 27 |
| 3.3 | Chapter Summary | 28 |
| 4 | Problem Analysis | 29 |
| 4.1 | The Case for L-Systems..... | 29 |
| 4.2 | Preliminary Testing, assumptions, and limitations..... | 30 |

| | | |
|----------|---|-----------|
| 4.3 | Tech Stack | 32 |
| 4.3.1 | SoundFont File Format | 32 |
| 4.3.2 | Flutter Framework | 32 |
| 4.3.3 | musicpy and sf2_loader Python libraries | 33 |
| 4.3.4 | Alternative Frameworks | 33 |
| 4.4 | Requirements Engineering | 34 |
| 4.4.1 | Functional Requirements | 34 |
| 4.4.2 | Non-Functional Requirements | 36 |
| 4.4.2.1 | Functionality | 36 |
| 4.4.2.2 | Usability | 36 |
| 4.4.2.3 | Reliability | 37 |
| 4.4.2.4 | Performance | 37 |
| 4.4.2.5 | Supportability | 37 |
| 4.4.2.6 | Plus | 38 |
| 4.5 | Chapter Summary | 38 |
| 5 | Lindenmayer Systems | 39 |
| 5.1 | Formal Definition | 39 |
| 5.2 | Types | 40 |
| 5.2.1 | Context-free (OL) | 40 |
| 5.2.2 | Context-sensitive (IL) | 42 |
| 5.2.3 | Stochastic (non-deterministic) | 43 |
| 5.2.4 | Deterministic (DL) | 44 |
| 5.2.5 | Propagative (PL) | 44 |
| 5.2.6 | Non-Propagative | 44 |
| 5.2.7 | Table L-Systems | 45 |
| 5.2.8 | Parametric | 46 |
| 5.2.9 | Applying Extensions on L-Systems | 47 |
| 5.2.10 | Using brackets to manage an L-System's interpretation state | 47 |
| 5.2.11 | Chosen L-System types for the development of <i>L-Music</i> | 47 |
| 5.3 | Chapter Summary | 48 |
| 6 | Design | 49 |
| 6.1 | System and Application Design | 49 |
| 6.1.1 | 4+1 View Model | 49 |
| 6.1.1.1 | Logical View | 50 |
| 6.1.1.2 | Deployment View | 50 |
| 6.1.1.3 | Implementation View | 51 |
| 6.1.1.4 | Process View | 51 |
| 6.1.2 | Application Design | 53 |
| 6.1.2.1 | User Interface Module | 53 |
| 6.1.2.2 | L-System Module | 53 |

| | | |
|----------|--|------------|
| 6.1.2.3 | Musical Parser Module | 55 |
| 6.2 | UI/UX Design | 56 |
| 6.2.1 | Color Scheme | 56 |
| 6.2.2 | Material Design | 58 |
| 6.2.3 | UX Concerns | 59 |
| 6.2.3.1 | Visual Hierarchy and Flow | 59 |
| 6.2.3.2 | Component Consistency | 61 |
| 6.2.3.3 | Creating a Custom L-System | 63 |
| 6.3 | Chapter Summary | 65 |
| 7 | Implementation | 67 |
| 7.1 | User Interface Module | 67 |
| 7.1.1 | Visual Hierarchy and Flow | 67 |
| 7.1.2 | <i>L-Music</i> UI Breakdown | 69 |
| 7.1.2.1 | SoundFont Instrument List Widget | 69 |
| 7.1.2.2 | L-System List Widget | 70 |
| 7.1.2.3 | Musical Parser List Widget | 71 |
| 7.1.2.4 | Musical Parser Parameters Section Widgets | 72 |
| 7.1.2.5 | Audio Manipulation Section Widgets | 75 |
| 7.1.3 | Conditioning User Input | 77 |
| 7.1.4 | Creating a Custom L-System | 78 |
| 7.2 | L-System Module | 83 |
| 7.2.1 | DOL-Systems (Deterministic, Context-Free) | 84 |
| 7.2.2 | DIL-Systems (Deterministic, Context-Sensitive) | 85 |
| 7.2.3 | Stochastic L-Systems (Non-Deterministic) | 87 |
| 7.2.4 | Table L-Systems | 89 |
| 7.2.5 | Considerations regarding L-System Implementation | 90 |
| 7.3 | Musical Parser Module | 90 |
| 7.3.1 | Showcasing the sf2_loader and musicpy libraries | 91 |
| 7.3.2 | Interpreting L-Systems Musically | 92 |
| 7.3.2.1 | Random Interpretation | 92 |
| 7.3.2.2 | Scale-Based Interpretation | 94 |
| 7.3.2.3 | Polyphonic Interpretation | 96 |
| 7.3.3 | Remarks on the generated musical fragments | 100 |
| 7.4 | Chapter Summary | 103 |
| 8 | Solution Assessment | 105 |
| 8.1 | Assessment Components | 105 |
| 8.2 | Tests | 105 |
| 8.3 | Results | 107 |

| | | |
|----------|-----------------------------------|------------|
| 8.3.1 | Usability Session | 107 |
| 8.3.2 | Hearing Session | 109 |
| 8.4 | Chapter Summary | 111 |
| 9 | Conclusions | 113 |
| 9.1 | Achieved Goals..... | 113 |
| 9.2 | Limitations and Future Work | 116 |
| 9.3 | Personal Remarks..... | 117 |

Figure Index

| | |
|---|----|
| Figure 1 – Example of sheet music (own authorship). In it, we can extract data such as key signature (indicated by the sharp sign at the right of the clefs), tempo (indicated in a loose way by the term <i>Allegro</i> and a more concrete way by stating that a whole note equals 120 Beats Per Minute (BPM)) and repetition structures (indicated by the two black lines and two dots right after the key signature). | 10 |
| Figure 2 – Examples of graphical interpretations of L-Systems (own authorship). | 12 |
| Figure 3 – A visual representation of a one-dimensional CA system. A transition rule of this cellular automata is the following: if a cell equals zero and both of its neighbors equal one, the cell will keep its value (zero) in the next time step. Extracted from [57], p.172. | 14 |
| Figure 4 – Value Proposition Canvas for the AMCS VSTi plugin | 27 |
| Figure 5 - Algorithm for a musical interpretation on an L-system | 30 |
| Figure 6 – Sierpinski Triangle generated from an L-System | 42 |
| Figure 7 - <i>L-Music</i> System Component Diagram..... | 50 |
| Figure 8 - <i>L-Music</i> Deployment View..... | 51 |
| Figure 9 - <i>L-Music</i> Package Diagram..... | 51 |
| Figure 10 - <i>L-Music</i> Sequence Diagram | 52 |
| Figure 11 - User Interface Module Component Diagram | 53 |
| Figure 12 – Object representation of a Lindenmayer System | 54 |
| Figure 13 - L-System Module Class Diagram | 54 |
| Figure 14 - Rough Sketch of <i>L-Music</i> 's initial landing screen | 60 |
| Figure 15 - Iteration on <i>L-Music</i> 's initial landing screen design..... | 61 |
| Figure 16 - Rough Sketch of <i>L-Music</i> 's Audio Manipulation Sliders..... | 62 |
| Figure 17 - Rough Sketch of the design for creating a custom L-System | 64 |
| Figure 18 - Empty Instrument List in <i>L-Music</i> | 68 |
| Figure 19 - Native Dialog to request a SoundFont File | 68 |
| Figure 20 - SoundFont Instrument List - Loaded State | 69 |
| Figure 21 - The L-System List. On the left, the selected item is in a collapsed state. On the right, it is in an expanded state, showing details about the "Sierpinski Triangle" L-System, namely its axiom and production rule set. On the top is an iteration counter for the user to indicate how many generations of the selected L-System are desired..... | 70 |
| Figure 22 - Touch Area differences between an Instrument List Item and an L-System List Item | 71 |
| Figure 23 - Musical Parsers List | 71 |
| Figure 24 - The top section of <i>L-Music</i> , consisting of the Instruments, L-Systems and Musical Parsers Lists..... | 72 |
| Figure 25 – Scale List | 72 |
| Figure 26 - Rhythm List..... | 73 |
| Figure 27 – Key Dropdown and Dropdown Menu | 73 |
| Figure 28 - Beats Per Minute Counter..... | 74 |
| Figure 29 - Root Pitch Dropdown and Dropdown Menu | 74 |

| | |
|--|----|
| Figure 30 - Number of Octaves Counter | 75 |
| Figure 31 - The Generate Music Button. On the left, it is in its active state. On the right it is in a disabled state, showing a progress indicator on the right to inform the user that the operation is still being performed..... | 75 |
| Figure 32 - Parser Parameters Section. | 75 |
| Figure 33 - Audio Playback Buttons..... | 76 |
| Figure 34 - Save audio as WAV or MIDI File buttons | 76 |
| Figure 35 - Example of a Snackbar in <i>L-Music</i> . In this case, the system is informing the user that a file named "sample_test.wav" was saved successfully in the user's device. | 76 |
| Figure 36 - <i>L-Music</i> Audio Sliders. In order, from top to bottom: audio position slider, audio volume slider and audio playback speed slider. | 76 |
| Figure 37 - Component Layout of the Audio Widgets Set | 77 |
| Figure 38 - Example of conditioning user input. All parser parameters are opaque given that the selected parser is "Random" | 78 |
| Figure 39 - The stacked layout of <i>L-Music's interface</i> | 79 |
| Figure 40 - L-System type Dropdown and Dropdown Menu | 80 |
| Figure 41 - Text Fields for creating an L-System in <i>L-Music</i> | 80 |
| Figure 42 - Error Message for a mal-formatted L-System alphabet when creating an L-System | 81 |
| Figure 43 - Error Message for an invalid axiom when creating an L-System. In this case the defined alphabet only contains the symbols "A" and "F" | 81 |
| Figure 44 - Error Message for an invalid production rule when creating an L-System | 81 |
| Figure 45 - Empty rule set list view when creating an L-System..... | 82 |
| Figure 46 - Rule set list view after a rule is added | 82 |
| Figure 47 - Example of a custom L-System list item | 83 |

Table Index

| | |
|--|-----|
| Table 1- Functional Requirements of the <i>L-Music</i> application | 34 |
| Table 2 - <i>L-Music</i> 's Color Scheme | 57 |
| Table 3 - Musical Parser Configurability. "NC" stands for Non-Configurable. "C" stands for Configurable. Row names represent Musical Parser Types. Column names represent parser parameters..... | 77 |
| Table 4 - Collected data from participants prior to beginning test session | 106 |
| Table 5 - Generated Excerpts from <i>L-Music</i> that were used for a hearing session..... | 109 |
| Table 6 - Results from hearing session to determine if generated excerpts from <i>L-Music</i> could be judged as human composed | 110 |
| Table 7 - State of defined Use Cases at the end of the project | 114 |
| Table 8 - State of defined non-functional requirements at the end of the project | 115 |

Code Snippet Index

| | |
|---|----|
| Code Snippet 1- Base classes for the implementation of the L-System module | 84 |
| Code Snippet 2 - Implementation of a DOL-System | 85 |
| Code Snippet 3 - Implementation of a DIL production rule..... | 86 |
| Code Snippet 4 – Implementation of a DIL System | 86 |
| Code Snippet 5 - Implementation of a Stochastic production rule..... | 87 |
| Code Snippet 6 - Example of a StochasticRule object | 87 |
| Code Snippet 7 - Implementation of a Stochastic L-System | 88 |
| Code Snippet 8 - Example of a Production Rule Table | 89 |
| Code Snippet 9 - Implementation of a Table L-System | 90 |
| Code Snippet 10 - <i>sf2_loader</i> library usage example | 91 |
| Code Snippet 11 - <i>musicpy</i> library usage example | 91 |
| Code Snippet 12 - Implementation of the random musical interpretation of an L-System..... | 93 |
| Code Snippet 13 - Implementation of the scale-based interpretation of an L-System..... | 95 |
| Code Snippet 14 - First Implementation of a Polyphonic interpretation of an L-System, relying only on “F” symbols | 97 |
| Code Snippet 15 - Polyphonic Implementation using "F" and "G" symbols | 99 |

Acronyms

Acronyms

| | |
|-------------|--------------------------------------|
| AI | Artificial Intelligence |
| ANN | Artificial Neural Networks |
| CA | Cellular Automata |
| CNN | Convolutional Neural Network |
| DAW | Digital Audio Workstation |
| GAN | Generative Adversarial Network |
| GRU | Gated Recurrent Unit |
| GUI | Graphical User Interface |
| HCI | Human-Computer Interaction |
| HMM | Hidden-Markov Model |
| LSTM | Long-Short Term Memory |
| MIDI | Musical Instrument Digital Interface |
| SI | Swarm Intelligence |
| VSTi | Virtual Studio Technology instrument |
| UI | User Interface |
| UX | User Experience |
| UWP | Universal Windows Platform |
| WAV | Waveform Audio File |

1 Introduction

This chapter serves as a brief introduction into the world of algorithmic music composition. First, a contextualization of music will be given. This will be followed by a brief overview on the history of algorithmic composition research. Then, the concepts of autonomous and assisted music composition systems will be presented. The problem this thesis tackles and intends to contribute a valuable input to and its goals will be showcased next. Finally, the outline of the document will be introduced.

1.1 Context

Throughout history, the presence of music in a society can always be found. Even though what it will sound like, be used for and the ideas or emotions that it conveys will vary from culture to culture, it is an art form that is considered universal [1]. There is a circular loop that music has with culture, meaning that it feeds from it to evolve but also becomes a driving force to change it at the same time (e.g. the punk movement was strongly connected to the style of music of the same name) [2]. Music also plays an important role in the stages of an individual's life [3]–[5], their health [6] and the development of their own identity. Therefore, it is paramount to understand why and how music works. Notwithstanding, providing a formal definition for it has proven to be a complex problem due to its various facets [7]. Music is commonly defined as organized sounds, structured by a person, the composer, to an extent which will cause other people, the listeners, to react to the former. This definition, however, is incomplete because it is lacking the rationalization of how those organized sounds were arranged. To construe what music is, cultural, functional, operational, structural, and personal aspects should be acknowledged. We can then characterize music as organized sounds that were influenced by external and internal (meaning ideas that come from the composer's own identity) aspects [8]. It is also possible to think of music as a language, since it has a way to be represented in writing, a way to be transmitted through sound and its own set of grammatical rules. These will change from culture to culture, just like any type of language [9]. It is also of the essence to acknowledge the influence that technology brings to the development of music,

most notably in the 20th and 21st centuries. With the creation of electronic instruments, music related software and new processes for recording, composing, and performing, music has branched out into very distinct areas (if we compare the lexicon of electronic music and western classical music, similarities may be found, but the end result is completely different because of the divergences of the musical styles).

1.2 Algorithmic music composition

Algorithmic music composition is a formal process, and an extensively researched field, which aims to create music with the least human intervention possible. There are several techniques which can be employed in order to create musical pieces, ranging from stochastic processes to rule-based systems to Artificial Intelligence (AI) [10]. Even though the term appeared in the 20th century, there were already methods to create music in an algorithmic fashion before that.

In the late 15th century, canonic composition was created where a single voice part was composed. Then, rules, or canons, were written out for additional voices based on that single voice part. As an example, the second voice could start singing the same melody after several measures of the first voice's start. The same canon would apply to the third and fourth voices and from a simple rule, the resulting piece would be richer in texture [9][p. 166]. In 18th century Europe, a popular dice game called *Musikalisches Würfespiel*, German for "musical dice game", was appealing to composers like Mozart, who ended up developing his own version. For this game, the order of pre-written musical fragments from a set was determined by the results of rolled die, i.e., a stochastic process in which probability determines the output. The ordered fragments could then be performed to hear a new composition [11].

With the appearance of computers, algorithmic composition evolved rapidly during the 20th century and into the 21st. Lejaren Hiller and Iannis Xenakis made significant contributions to this growth [12]. Hiller is credited for creating the first computer-composed piece of music, *The Illiac Suite for String Quartet*, in 1956 [13]. The suite is made up of four movements, each one composed out of increasingly complex algorithms. All the algorithms relied on probabilistic methods, such as the Monte Carlo method or Markov Chains, and a set of music composition rules defined by Hiller. Compositional elements such as first-species counterpoint, four-part voicings, dynamics and varying rhythm were successfully implemented in the algorithms [14]. Despite having some flaws such as the lack of a common theme throughout the movements, the *Illiad Suite for String Quartet* became a notable example in the field of algorithmic composition [13]. Iannis Xenakis is regarded as one of the important composers of 20th century classical music. He is considered a pioneer in applying mathematical models to musical composition and for also integrating concepts from other fields, like architecture, into the realm of music. Like Hiller, Xenakis also relied on probabilistic methods like Markov chains, but also used models from set theory and game theory. His repertoire is considered to be very raw and energetic [15]. Xenakis' contributions to algorithmic composition stem from several works: (i) his book, *Formalized Music: Thought and Mathematics in Composition*, where he

describes his *Markovian Stochastic Music* in detail; (ii) a computer system called UPIC, which could transform graphical images into musical sounds; (iii) a stochastic synthesizer algorithm which used Digital Signal Processing (DSP) techniques to generate random timbres and musical forms and (iv) an autonomous composition program called the *Stochastic Music Program* (SMP), which relied on stochastic formulas to generate a complete score written for conventional instruments. There were also several programming languages and environments developed near the end of the 20th century which are still used today for real time audio synthesis and algorithmic composition. Notable examples include SuperCollider and Max/MSP [12].

With this overview on the history of algorithmic composition, we can conclude that the concept is not new. Furthermore, one can argue that composers regularly apply algorithms, in an unconscious way, when they are involved in the process of writing their music. If we consider music to be the structuring and organization of sound, it is possible to encapsulate these structures into formal processes. Consequently, a new question arises from having the means to implement software that can write music: should the process be automated? Or should composition algorithms be used in a collaborative fashion with the human composer?

1.2.1 Autonomous and Assisted Music Composition

Autonomous and assisted music composition serve different goals and have different definitions. Therefore, it is essential to distinguish them and understand their differences. Autonomous music composition assumes that the implemented program is able to compose music without any input from the composer. It has the innate capability of writing a fully-fledged musical piece, using whatever mechanisms were initially implemented by the programmer who wrote it. A notable consequence of automated music composition is that it removes the composer from the process completely after the program has been implemented. In fact, the composer might not be even part of the process if the mechanism in which the program relies upon to write music does not need a pre-defined musical rule set. On the other side of the spectrum, assisted music composition exists to create or augment synergy between composer and machine. It should be seen as a commensalism relationship, where the composer can give input or ask for feedback to the computer, and then benefit from its responses during the creative process. In Chapter 2, examples of both approaches to musical composition are reviewed.

1.2.2 On the pursuit of computer-generated art

Computer-generated art raises questions around authenticity, autonomy, computational creativity, and resistance to change. While the first three traits are connected with each other and inside the same scope (artistic integrity and art appreciation), resistance to change is, in this case, approached under a different context, that of which is the history of art and art making.

Creativity affects our perception of an artwork. Therefore, we must define it. Creativity's standard definition is bipartite: for something to be creative, it has to be original and effective. By original, we mean something that is novel, out of the ordinary or distinctive. By effective, we mean that which brings value. This value can be created by (i) evoking emotions or realizations on an audience or individual, (ii) an idea being accepted as innovative and relevant by the respective community or (iii) by a product that is useful to its consumers [16]. There are extensions to this definition. Bogden adds a third factor to something being considered creative, which is the element of surprise [17]. She further argues for three types of creativity: combinational creativity, where new ideas are generated by combining already existing ones; exploratory creativity, which results from developing novel and unexpected ideas within structured spaces¹, and transformational creativity, where new dimensions are created and the space itself is changed, resulting in a much bigger surprise than ideas originated by combinational or exploratory creativity [17].

Classifying the creativity of a computer system as creative has proven to be a topic worthy of discussion [18]. If we consider the standard definition of creativity, it is possible that a computer system be creative. Generating novel output is simply a matter of integrating different inputs such as to obtain a never-before-seen combination. The generation of value, however, is where computer-generated art struggles most [19]. We first need to consider, nevertheless, that different contexts under which creativity is being evaluated will state how that value is retrieved. This means that a given system can be creative in playing a game but might not be creative at writing a piece of music, because of how different value is extracted from both worlds [20]. As an example of creativity in playing a game, move 37 of DeepMind's algorithm AlphaGo against Lee Sedol, a Go champion, is a notable example. AlphaGo played a move that a human player would most likely not perform because it would be seen, due to the prejudice of the history of Go and its strategies being passed on from generation of players to the next, as a bad tactic. Nevertheless, it was a crucial part of AlphaGo's strategy to win the second game against Lee Sedol. This move was remarked as innovative, changing the way Go players looked at the game² [19], [20], [22]. AlphaGo was capable of Bogden's transformational creativity. In spite of this, AlphaGo was solving a problem (calculating where to place its pieces in order to beat its opponent based on a set of rules). In the context of art, computer systems are not trying to solve problems, and therefore it is harder to assess how value is brought forth [19]. Typically, this is done by having experts of the art domain interacting with the system (e.g., musicians interacting with a system that can play music alongside them) or evaluating the reception of the produced art with case studies (e.g., evaluating whether

¹ A space is a set of dimensions with its own culture and rules. As an example, a given genre of music is a space, where musicians can develop their own dimensions and potentially transform the space itself, thereby pushing it forward [17].

² Sometimes experts of the art domain do not recognize the artistic value of their contemporaries in an immediate fashion. Case in point, some works of music of Johann Sebastian Bach were only recognized its merit long after the composer had passed away [21].

the system's output evoked any emotion on its receptors) [19], [23]. The Lovelace test, an improvement upon the Turing test that relies on proving a restrictive epistemic relation between an artificial agent and the human who designed it, was also proposed as a way of evaluating if a computer system possesses creativity. It states that a computer system C and its output O, designed by a human agent H, possesses creativity, if and only if human agent A cannot, in any way possible, explain how C came up with O [24].

J. McCormack et al. argue that the creativity of generative art systems has been exaggerated due to the general public's lack of understanding of how advanced these systems really are. They reason these do not possess the same type of autonomy or intention as a human artist [25]. Whereas a human artist intentionally creates art for myriad reasons, a computer program is built for that purpose and does not possess the free-will or the intention of creating art. This difference ends up affecting appreciation of computer-generated art due to how we perceive and give value to art works. Studies conducted by Chamberlain et al. also verify the latter observations [23]. In the first study, the ability to detect computer-generated and man-made art (in this case, paintings) was reviewed as well as how categorizing the art works affected their perceived value. A bias towards computer-generated art was found. What the participants perceived to be man-made ended up being appreciated more. The second study sought to discover what created that bias. The conclusions were that computer-generated art was appreciated more when the participants were able to see an anthropomorphic robotic agent creating the art, rather than just showing the finished product for assessment [23]. This suggests the same idea of artistic intention stated by J. McCormack et al. Seeing and understanding the creation process gave the participants insight into the artificial mind of the robotic agent. Additionally, the participants did not find the robotic agent's works creative [25]. Notwithstanding, both authors recognize the value and potential in human artists utilizing these tools to augment their artistic process.

There seems to be a consensus towards defining creativity for computer systems that generate art. Not only does it need to be original, create value and be surprising, as creativity is commonly defined, but the computer system itself must intentionally create art [17], [19], [23]–[25]. It is not enough for it to be programmed by someone to engage in the creative process. Furthermore, current artificially intelligent systems, while possessing capabilities to be truly original and to create value, lack intentionality and as such cannot be considered equally creative to a human. Whether to consider these systems as creative or not will depend on the definition of creativity that the person upholds as the correct one. In spite of this, it is necessary to acknowledge that these systems sometimes show that humans are not thinking beyond the boundaries of their domain, and as such, it creates new expectations about what is possible to do inside it [19].

When something new arises, it is always met by resistance to change, be it at the community level or at the individual level. Feelings of insecurity for the unknown, the force of habit, or selective perception and retention are but a few reasons for someone to attempt to stop the process of transformation. Resistance to change can be broken down

to four stages, where (a) the first stage tackles the appearance of the innovation and its adoption by a minority; (b) the second stage deals with the movement towards change begins to grow and the pro and con parties are defined; (c) the third stage marks conflicts between the former groups and it is where the success of resisting the change is decided and (d) the fourth stage, where, if successful, the change is supported by the majority and those who resist it begin to be ignored (albeit the change can still be resisted to if proper cautions are not taken into account) [26]. Although resistance to change is mainly studied under the context of organizational behavior [27], examples of this phenomenon can be found in other domains. With regards to music, there are some notable examples where resistance to change took place. Moog synthesizers were temporarily banned from use in commercial work in the USA (United States of America), out of a fear (which was later verified) that synthesizers would put session musicians out of work. The ban was later lifted, and the Moog synthesizers had a great impact in the way musicians would tackle writing music. They were also considerably important in raising the popularity and interest in modular and analog synthesizers [28]. Rock music is another example of resistance to change, although more so from the consumer's perspective. Due to the nature of the subjects that the style of music encompasses and to its sonic aspects (particularly the associated loudness that is connected to the use of amplified electronic instruments), rock music was targeted for long periods of time over the matter of moral concerns [29]. The same can be illustrated by rap music [30]. Even so, through a slow acceptance of each, both genres are regarded as art forms and have an established position within the music world [31], [32]. With these examples, we can assume that when new tools, especially ones that rely on artificially intelligent systems, that aid musicians in their creative process become usable and popular, there will be a movement to resist them. We should strive to embrace them and realize their full potential, as they might possess the ability to push music making into new boundaries, like the Moog synthesizers once did in the 20th century.

1.3 Problem Description

Computer-generated music is a topic of interest, not only for its artistic value it brings by offering new ways of looking at sound and music, but also because of its technical complexity in structuring the levels of abstraction that are found in the former [10]. As mentioned in previous sections, systems that are capable of writing music were already created, using a myriad of techniques. These systems are designed to be autonomous or to be used by a composer during their creative process. However, most of these systems are very limited in what they can compose or are not readily available for users. They also lack integration with other music related software, such as notation programs like MuseScore or Digital Audio Workstations (DAWs) like Fruity Loops Studio.

The problem that we address in this thesis is the development of an assisted music composition system that can map musical properties to generate relevant and pleasant musical excerpts so that a composer can use them when writing music. As it is an assisted

music composition system, it will rely on user input to create the fragments that the composer desires. In other words, we are looking to answer the following questions:

- Can algorithmic music be of assistance to musicians or composers and augment their creativity? If so, what are the best set of techniques to achieve the latter? What interactions will users expect from such a system?

1.4 Goals

The development of the aforementioned system will support in answering the questions stated in 1.3. However, in order to achieve this, we must first:

- Study and review existing solutions in order to assess the underlying mechanisms and techniques that were implemented to achieve autonomous or assisted music generation. As examples, the DeepBach system relied on deep learning [33] while the Melomics system relied on formal grammars mixed with evolutionary and genetic algorithms [34].
- Understand how to model musical concepts and elements in a way that allows the system to be flexible and broad to maximize the number of available operations [34] (e.g. changing the dynamic from forte to piano at a given bar or changing time signatures at a given bar). This can be done, for example, by using Schenkerian Analysis [35].
- Design the interaction between composer and system in a way that will provide a good user experience for the composer when they are collaborating with it, namely when they are giving input to the system so it can generate the desired musical fragment.
- Implement the system using the most suitable technology stack. This will be determined by making some experiments and by the review of existing solutions.
- Establish a method to validate and assess whether the results are satisfactory or not. We interpret the results here as being the musical excerpts the system will output to the composer. One way to do this is would be by measuring the listeners' emotional response to those excerpts with the Discrete Emotions Questionnaire (DEQ) [36].
- Assess whether the system ends up being beneficial to its end users, the composers. This will require creating test sessions with a test group of composers.

1.5 Outline

The remaining content of this thesis is organized as follows.

Chapter 2 offers a study and review on state-of-the-art systems that offer autonomous and assisted music composition functionalities.

Chapter 3 encompasses the business value analysis on an envisioned product based on an assisted music composition system.

Chapter 4 analyzes the problem this thesis tackles and documents the chosen approach, as well as the developed prototype's tech stack and the definition of functional and non-functional requirements.

Chapter 5 introduces the reader to Lindenmayer Systems, or L-Systems, providing its formal definition as well as explaining multiple types of L-Systems.

Chapter 6 presents the system design of the developed prototype as well as the design of its user interface and user experience.

Chapter 7 documents the implementation of an assisted music composition system prototype named *L-Music*. The prototype is broken into three core modules, each with its own separate responsibility.

Chapter 8 reports the tests that were performed to assess the developed solution.

Chapter 9 provides the conclusions of this project, considering the achieved goals, limitations, and future work.

2 State of the art

This chapter offers the reader an extensive review of the state of the art for generative music systems. The outline of the chapter, based on a recent survey [10], follows. First, a brief discussion on musical information abstraction, and how it has an impact on the system's design as well as interaction in case a graphical interface is being developed. After, an overview of nonadaptive and adaptive algorithmic music composition systems. The chapter ends with a review of notable examples of both nonadaptive and adaptive music generation systems.

2.1 Musical Information Abstraction

One of the key issues in developing music composition programs is the abstraction of musical information. Humans themselves already perform some abstraction when actively listening to music (e.g., identifying if a certain set of sounds are happy or sad) [37], [38]. Modelling systems to capture manifold abstract concepts (e.g., dynamics, melody, or harmony) related to Western music theory has proven to be one of the most interesting challenges of this research area [10]. Moreover, in the event that a Graphical User Interface (GUI) is also being developed, one must also dedicate their attention into designing a proper Human-Computer Interaction (HCI) that (1) reflects the systems' abstraction capabilities and flexibility, and (2) allows the composers to intuitively translate their own perceptions of these abstract concepts into the system to maximize synergy between both parties, therefore augmenting the composer's creativity [39].

Abstracting musical information is not a new topic. The musical notation system that is used for western music theory already achieves abstraction to some degree, providing a visual concretization of musical ideas, such as key signatures, repetition structures or pitch and rhythm information. Figure 1 shows an example of the former. In algorithmic music composition, a procedural nature can be found in the representation models, i.e., the composer describes to the algorithm how it should generate its output through parametrization. Duignan also points out to resource and instance models. Resource models

rely on various objects that are updated over time, whereas instance models are created per musical event [39]. Designing a generative music system to use one type or the other will have consequences regarding the system's abstraction capabilities and its flexibility.



Figure 1 – Example of sheet music (own authorship). In it, we can extract data such as key signature (indicated by the sharp sign at the right of the clefs), tempo (indicated in a loose way by the term *Allegro* and a more concrete way by stating that a whole note equals 120 Beats Per Minute (BPM)) and repetition structures (indicated by the two black lines and two dots right after the key signature).

Some musical concepts, such as pitch class and tonality, have been approached with accuracy with geometrical information techniques. This allows for a compact representation of musical data which proves more useful for analysis purposes than music generation ones [10]. Although it is more useful from the perspective of the former, it indicates a possible approach for existing generation techniques to draw upon to improve their own abstraction methods. In the following sections, we will further review how different music generation techniques tackle abstracting all the different hierarchies and perspectives found within music.

2.2 Nonadaptive Algorithmic Music Composition

Nonadaptive music generative systems heavily rely on the musical knowledge of the person who creates them. As the name suggests, these systems do not possess the capability of changing themselves towards possible improvements as they generate more and more output. In order for nonadaptive systems to be successful at generating interesting pieces of music, a proper mapping between musical knowledge to symbolic data must be attained. Musical rules are explicitly linked to a given output of the system. Therefore, the design and development of the mapping itself is what ends up affecting the output more [10]. We will discuss three methods that have been used extensively to algorithmically write music: Lindenmayer Systems (L-Systems), Cellular Automata (CA) and Swarm Intelligence (SI).

2.2.1 Lindenmayer Systems

Lindenmayer Systems (L-Systems) were initially proposed by Aristid Lindenmayer as a way to mathematically model plant development. At their core, L-Systems are parallel generative grammars. L-Systems then vary according to the type of production rules they employ. The simplest type of L-Systems, which are deterministic and context free, or DOL-systems, can be formally defined as follows [40]:

“Let V denote an alphabet, V^ the set of all words over V , and V^+ the set of all nonempty words over V . A string OL-system is an ordered triplet $G = \{V, \omega, P\}$ where V is the alphabet of the system, $\omega \in V^+$ is a nonempty word called the axiom and $P \subset V \times V^*$ is a finite set of productions. A production $(a, \chi) \in P$ is written as $a \rightarrow \chi$. The letter a and the word χ are called the predecessor and the successor of this production, respectively. It is assumed that for any letter $a \in V$, there is at least one word $\chi \in V^*$ such that $a \rightarrow \chi$. If no production is explicitly specified for a given predecessor $a \in V$, the identity production $a \rightarrow a$ is assumed to belong to the set of productions P . An OL-system is deterministic (noted DOL-system) if and only if for each $a \in V$ there is exactly one $\chi \in V^*$ such that $a \rightarrow \chi$.” [40], p.4.*

To showcase this definition, a DOL-system is presented below:

Alphabet V is comprised of the letters $\{a,b\}$.

Let the production rules be:

$a \rightarrow ab$

$b \rightarrow a$

If we consider our axiom as b , then we get the following first five iterations:

- Iteration 0: “b”
- Iteration 1: “a”
- Iteration 2: “ab”
- Iteration 3: “aba”
- Iteration 4: “abaab”
- Iteration 5: “abaababa”

If we take the length of each production string, we can verify the existence of the Fibonacci Sequence. Due to their origins, L-Systems were rapidly translated to a graphical interpretation, using what is known as turtle graphics, based on the LOGO programming language [41]. The procedure for this is to map letters of the L-System’s alphabet to graphical movements that will be executed by the turtle, consequently altering its state, once all iterations are ran. The state of the turtle is represented by the triplet (x,y,α) , where (x,y) represents the turtle’s Cartesian coordinates and α denotes the turtle’s heading, i.e., the direction in which it is heading. Typically, the following symbols are used in alphabets [40], [41]:

- F, move the turtle forward by a length of d and draw a line between (x,y) and (x',y') .

- f , move the turtle forward by a length of d without drawing a line.
- $+$, turn the turtle left by angle $+\delta$.
- $-$, turn the turtle right by angle $-\delta$.
- $[$, save the turtle's current state.
- $]$, pop the turtle's current state.

Even with simple production rules, one can obtain rather interesting and visually appealing drawings resembling plants or leaves. It is also possible to draw fractal shapes. Figure 2 showcases examples of this.



Figure 2 – Examples of graphical interpretations of L-Systems (own authorship).

On the left: A Dragon Curve, a self-similar fractal shape. Number of Iterations: 15. Axiom: FX .

Production Rules: $X \rightarrow X+YF+$, $Y \rightarrow -FX-Y$. $\delta = 90$ degrees.

On the right: a bush-like plant. Number of Iterations: 5. Axiom: $++++F$. Production Rules: $F \rightarrow$

$FF-[-F+F+F]+[+F-F-F]$. $\delta = 22.5$ degrees.

Graphical renderings of L-Systems have been extended to three dimensions, as well as having evolved to simulate a realistic growth and modelling of plants, flowers, and trees [42]–[44]. Musical interpretations of L-Systems have been thoroughly researched [15], [34], [35], [45]–[51]. The first publication regarding music generation via L-Systems was authored by Prusinkiewicz. The method relied on a direct mapping of the produced strings to a sequence of notes. The musical interpretation of Prusinkiewicz was tightly coupled to the graphical interpretation of L-Systems. By assuming the first note as the root of a given scale, (e.g., the C note is the root of the C Major scale) the pitch of the remaining notes would be decided by mapping the former to the turtle's y coordinate values throughout the graphical rendering [48]. McCormack developed a computer assisted music composition system that relied on non-deterministic L-System hierarchies and parametric extensions to generate MIDI (Musical Instrument Digital Interface) files. The interaction between McCormack's system and a

composer would be at the grammar injection level, i.e., the composer would have to write their own grammar, feed it to the system, and wait for the results in MIDI format. There are other attempts at mapping the graphical representation of the L-System into its musical counterpart [51]. Worth and Stepney structured their mapping methods in a way that would mirror Schenkerian Analysis³ onto their L-Systems, therefore implementing a Schenkerian rendering. Additionally, they explored stochastic and context-sensitive L-Systems to allow for more variation in rhythm and melody. Albeit having an additional challenge of producing pleasing visual and sounding interpretations simultaneously, some of the results attained by Worth and Stepney show that L-Systems can be an apt tool for generative music systems [35]. The generated excerpts can be listened to at the authors' publication page⁴. Melomics is an autonomous compositional system, based on evolutionary algorithms and L-Systems, developed by the GEB (*Grupo de Estudios en Biomimética*) of the university of Málaga. The results obtained by Melomics are very significant, going as far as having an orchestra performing its pieces, and showcases the capabilities that generative music systems can reach [34]. Section 2.4.1 discusses Melomics and as such will be skipped here. It should be noted that since Melomics employs evolutionary algorithms, it ceases to be a nonadaptive solution. Another relevant work is Stelios Manousakis' Musical L-Systems. Manousakis integrated L-Systems into electronic music by generating sounds based on the output of his L-Systems [15]. Section 2.4.3 discusses Manousakis' generative music and as such will be skipped here. Other works which employ variations on L-Systems or mix L-Systems with other techniques have also been proposed, these being evolving L-Systems' rules or grammars over time through evolutionary methods [10].

2.2.2 Cellular Automata

Cellular automata (CA), also known as tessellation automata or iterative circuit computers, consist of evolving parallel homogenous systems, which are discrete in time and space and are capable of displaying emergent complex behavior and cyclical patterns based on simple rules and configurations [10]. CA were first proposed by Jon von Neumann and Stanislaw Ulam [52] and have been popularized by Conway's Game of Life [53]. Much like L-Systems, CA have a wide range of applications, such as chemistry, biology, physics, and generative art [54]. CA systems can be thought of as one, two or three-dimensional infinite arrays, made up of atomic components, the finite automata (or cells), which will modify their state as time goes forward. At the beginning, it is assumed that all cells possess a copy of the same state. For each cell, we associate a neighborhood, i.e., an aggregate of adjacent cells. The rules for how a neighborhood should be defined may vary. For each time step, the cells will suffer state changes due to local interactions with their own neighborhood. These changes occur

³ Schenkerian Analysis consists of analyzing a piece of music and dividing into three parts, the middle ground, the foreground, and the background [35].

⁴ <https://www-users.cs.york.ac.uk/~susan/bib/ss/nonstd/eurogp05/index.html> (Accessed February 20, 2021).

according to a set of transition rules that consider all of a neighborhood's cells' states [55], [56]. Manipulation of the transition rules, neighborhood definition, and other factors are what allow CA systems to be applied in diverse areas to achieve myriad results [10]. Figure 3 illustrates an example of a one-dimensional system. Although it is simple, the emerging complexity that it originates is visible in its iterations.

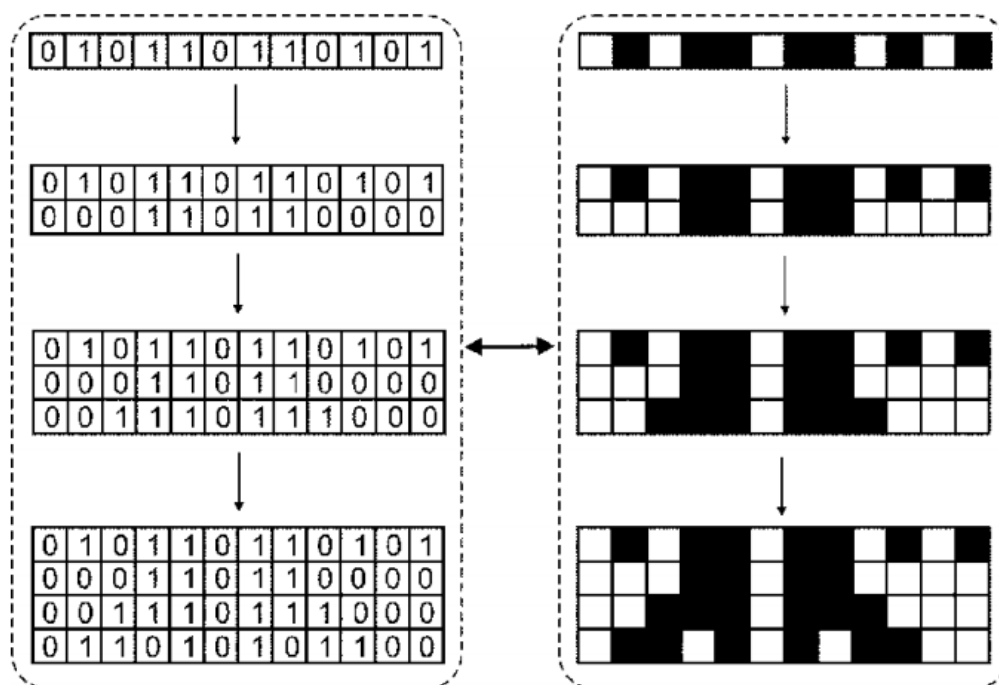


Figure 3 – A visual representation of a one-dimensional CA system. A transition rule of this cellular automata is the following: if a cell equals zero and both of its neighbors equal one, the cell will keep its value (zero) in the next time step. Extracted from [57], p.172.

When it comes to music generation, cellular automata which are cyclic, self-organized and show pattern properties are the ones to create more interest [58]. This is due to the referred traits also being found throughout music and musical compositional processes [57]. Notwithstanding, other cellular automata have also captivated the interest of music generation researchers, due to their diversity of behavior [10]. To achieve music generation, several symbolic mappings have been employed [59]. These include assigning areas of a CA space to notes on a musical score [60] and a graphical application that allows the composer to see a graphical rendering of the CA while also changing parameters related to its musical counterpart such as pitch and tempo. In this program, the composer could also interact directly with the visual representation of the cellular automata which would propagate changes to the MIDI notes being played [61]. Audio synthesis through granular synthesis was also applied to CA to generate sound. This was achieved by having each cellular automaton represent a granule (granules are short-timed sounds which are then put together to form a larger sound event, i.e., granular synthesis). The generation of these granules was based on emulating electric current flow on each of the CA's cells [58].

2.2.3 Swarm Intelligence

Swarm intelligence (SI) describes the behavior of individual particles that are part of a system. Inspired by biological phenomena such as swarming, flocking, and foraging, SI algorithms rely on the emerging behavior that results from the location of each individual particle that is part of the swarm [62]. This location is updated according to the velocity of the particle as well as the velocity of its neighbors. Several implementations of swarm particle algorithms exist, such as particle swarm optimization (PSO) and ant colony optimization (APO) [62]. SI can be applied in a wide array of fields. Examples of the latter include the simulation of swarms of humans or animals in movies [10], data mining and job scheduling [63]. Swarm approaches have also been researched for generative music systems [10]. By showcasing the similarities between the behavior and organization of swarms and musical ideas and concepts, Blackwell developed a system capable of mapping swarm particle positions to MIDI events, SWARMUSIC [64]. An interesting characteristic of Blackwell's system is that it does not attempt to encode musical knowledge, making it an improvisational system rather than a compositional one. This was achieved, in part, by removing global and neighborhood bests that are commonly found in swarm optimization algorithms. The musical mapping relied on using the swarm's particles position in a n-dimensional space to extract values for properties that could be defined in MIDI format (pitch, loudness, and pulse). After, the MIDI information can be transferred to any synthesizer to be able to be played. Of the experiments Blackwell put together, one included an improvisation between a human singer and SWARMUSIC, with a very positive outcome. The singer was able to adapt and cooperate with SWARMUSIC and create interesting melodies and harmonies [64]. This system was further extended to support multiple swarms interacting with one another to improvise music [65]. Additions to the system were collision avoidance and operators to allow a more fine-grained control of the swarms' outputs for performances (e.g., prohibiting the swarms to go outside of a certain key). Results were positive as well, with SWARMUSIC even participating with musicians at live concerts. This showcases the interactivity that generative music systems can achieve, in real-time, while also demonstrating the self-organization and similarity patterns that one can find in music [65]. Other applications of swarm algorithms to generate music were conceived towards an art installation [66], simulating ant movements to generate melodies [67], generic libraries to allow swarm-based music generation [68], and mixing swarms with genetic algorithms to compose traditional Chinese music [69], [70].

2.3 Adaptive Algorithmic Music Composition

Adaptive generative music systems have an advantage to nonadaptive methodologies when attempting to emulate musical styles. Due to their statistical learning of musical patterns, which can be implicit or explicit, adaptive systems do not require the person designing it to be an expert in the field of music. We discuss several methods that have been used to adaptively generate music: explicit modeling, implicit learning, evolutionary approaches, conceptual blending and deep learning [10].

2.3.1 Explicit Modeling

Probabilistic techniques allow to capture the frequency in which specific elements of a dataset happen. When applied to music, these explicit elements range from notes in a melody to chords in a progression. Higher level concepts can also be captured, such as cadences⁵. Because of this, emulating a given musical style is more attainable than resorting to rule-based modeling (i.e., nonadaptive methods) [10]. Also, these methods are often employed in musical analysis and musical information retrieval [72]–[74].

Hidden Markov Models (HMMs) are one of the most used statistical models to achieve the latter [10]. HMMs can be summarized by the following triplet: $\lambda = (A, B, \pi)$, where A is the state transition probability distribution, B is the observation symbol probability distribution in a given state and π is the initial state distribution. Unlike a non-hidden Markov model, where a sequence of symbols that is output by it can be directly related to the probability functions of the model's state machine, HMMs rely on stochastic processes that attempt to explain how a sequence of symbols came to be [75]. Common examples to demonstrate HMMs include a coin tossing scenario where the person flipping the coin(s) cannot be seen or drawing colored balls from multiple urns. In each, the goal is to estimate, through an HMM how the sequence of coin sides and drawn balls is being determined [76], [77]. HMMs have been used to harmonize chorales in the style of J.S. Bach. Conveying the same logic that is traditionally employed to compose a chorale (starting from a melody line in the leading voice, i.e., soprano, and harmonizing the remaining three voices according to that melody, i.e., alto, tenor, and bass), Allan and Williams devised an HMM that would attempt to predict what harmonies (hidden states) to write for given melody lines (visible states). Furthermore, they imposed a second HMM to write ornamentations (hidden states) for the melody line based on the harmonies the first HMM would suggest (visible states) [71]. Paiement et al. translated an HMM to a graphical model to generate chord progressions. One interesting choice in their approach was to not rely on Western music theory notation of chords (e.g., "minor", "major", "augmented") but rather choose to consider psychoacoustic properties of chords, such as perceived closeness [78]. Paiement further developed this work, integrating machine-learning (ML) techniques with HMMs and Input Output Hidden Markov Models (IOHMMs) with the goal of statistically modelling higher-level musical concepts, such as harmonization, rhythms, and melodies [79]. Other types of Markov Models have been used to generatively write music [80], [81]. Anderson et al. developed GEDMAS (Generative Electronic Dance Music Algorithmic System), an autonomous generative music system that created Electronic Dance Music (EDM) compositions based on probabilistic and first order Markov chain models. Much like the aforementioned systems, GEDMAS was trained on a specific corpus of EDM to learn how to extract information to use in its generation process. GEDMAS' compositions are musically interesting,

⁵ In Western music theory, a cadence is a melodic or harmonic idea that creates a sense of resolution and resolve. It is typically used to conclude a piece of music or a section of it [71].

emulating the learned styles in an appropriate fashion⁶ [80]. Continuator, a real-time interactive generative music system developed with constrained Markov Chains [81] is discussed in section 2.4.2.

Brown developed a generative music system for live performance which created musical scores in real-time for musicians to sight read and perform. The system initially relied on CA to generate musical data (i.e., pitch and rhythm patterns). However, random walk⁷ processes were chosen in the end to map the musical material. CA mappings were too uneven to be comfortable for sight-reading, while random walk processes fixed the issue while offering the same level of texture variation as CA. Gaussian distributions and selections were used to constrain the use of random walks when generating pitches and rhythms [83].

Roig et al. developed an autonomous composition system capable of generating original excerpts in a given style based on a database that held musical information extracted from musical information retrieval and analysis. The latter was performed via several statistical methods applied to MIDI data sets and divided into three segments to estimate given properties of the data sets: (i) tempo estimation through Inter-Onset-Interval (IOI) analysis between two notes and temporal fine adjustment, (ii) time signature estimation through a Rhythm Self-Similarity Matrix (RSSM) and (iii) rhythm and pitch contour extraction (extracted from MIDI event data and generated information from the previous steps). The autonomous composition system then relied on the database's information and some user selectable parameters (e.g., time signature, tempo, instrument) to generate musical excerpts⁸. The results were quite positive, considering that the system's excerpts were hard to distinguish from human composed ones [84].

2.3.2 Implicit Learning

Implicit learning allows for a compressed extraction of musical data into latent spaces. It is a more abstract approach than explicit modeling because it is not always known what parameters are contributing most towards preferable results. Artificial Neural Networks (ANNs) and deep learning are the main contributors for implicit learning music generation systems [10]. ANNs can be seen as a graph with interconnected (weighted edges) layered nodes. ANNs possess one or more input nodes and one or more output nodes (i.e., input layer and output layer). Any nodes between the latter layers are considered to be in hidden layers. The neural net is fed data from a training dataset through the input layer, which is processed considering the edges' weights and pre-defined functions (activation functions) as it goes through the net's hidden layers. This is where the network learns how to represent the

⁶ GEDMAS' music can be listened to at <https://soundcloud.com/pitter-pattr> (Accessed March 1, 2021).

⁷ A random walk is a stochastic process consisting of a set of successive steps in a mathematical space [82].

⁸ Samples from this system can be listened at <http://www.atic.uma.es/composition/> and <http://www.atic.uma.es/stylereplication/> (Both accessed March 4, 2021).

original data. At the output nodes, the learned features are used to generate predictions of results of the employed training dataset for comparison purposes. If the defined comparison criteria are not satisfied, the neural net will resort to backpropagation to tweak the edges' weights in order to attempt to generate more acceptable results [85]. A neural network is classified as deep if it has more than one hidden layer between the input and output layers [86]. Recurrent Neural Networks (RNNs) are among the most utilized nets for music generation purposes because they have the ability to retain data in a local memory [10]. A traditional RNN allows outputs to be fed as inputs and computes an output per timestep, therefore creating a feedback loop. For music generation, RNNs typically use a one-to-many structure, meaning one input node and multiple output nodes [87]. Much like explicit modeling, melody generation, harmonization (chord progressions) and musical style emulation have been researched and successfully implemented using neural networks [10]. Initial efforts with RNNs, such as the one by Mozer and Soukup, were successful in generating short, styled melodies, but lacked in adapting the networks to retain a sense of a higher, long-term structure (e.g., musical patterns). This is due to the RNN's feedback loop, which causes the neural network to retain information that might be too old in its local memory [88]. To fix this, long short-term memory (LSTM) networks were proposed. LSTM networks bring the concept of forget gates into play, which allows the neural net to choose relevant information from the past while forgetting any data that is of the opposing quality, therefore solving long-term dependency problems of a traditional RNN. Forget gates are placed on the hidden layers of the network, at each node. One of the first successes with LSTM networks was in 12-bar blues generation, with the network achieving knowledge of the style's structure from beginning to end [87]. Bach Chorales composition has also been recently achieved successfully when resorting to LSTM networks. BachBot, developed by Liang et al., encoded polyphony sequentially and restricted the information to pitch and rhythm only, achieving a compressed vocabulary to represent complex harmonies. Achieved results were positive, since the network's chorales were distinguished from the original material at a low rate from a large group of tests [89]. Sturm et al. tackled folk music transcription and generation with a deep generative LSTM network [90]. Musical information mapping was solved by resorting to the ABC notation [91] to create two distinct symbolic vocabularies, one based on characters and the other based on tokens. Two neural nets were trained, one for each defined vocabulary. Both networks' results were overall positive⁹. The produced content showed the capability of the networks to identify and apply higher-level concepts such as repetition, contour and resolution that are important in musical composition. Besides assessing autonomous composition, Sturm et al. also employed the networks as an assisted music composition tool, achieving positive results as well even when the network was fed information from a composer that varied greatly from its training material [90], [92]. Notwithstanding, LSTM RNNs are computationally expensive [10]. Further research on music generation with RNNs was made by [93], [94] using Gated Recurrent Units (GRUs). GRUs behave in a similar fashion as LSTM's forget gates but are less expensive due to not having an output gate and thus possessing fewer parameters to be evaluated and tweaked [93]. Colombo et al. developed

⁹ Publicly available at <https://soundcloud.com/oconailfamilyandfriends> (Accessed March 6, 2021).

BachProp, a deep GRU RNN to generate fully fledged musical pieces in a given style (as the name suggests, Bach chorales is one of them) [94]. Musical data mapping was achieved in a simple way, from MIDI notes to the network's representation of the former, by assuming that each note was a triplet, consisting of the note's pitch, duration, and time-shift relative to the previous note in the song. BachProp's network functioned similarly to the aforementioned RNNs, with its responsibility being that of probabilistically predicting the next note in a given sequence of notes. The achieved results were positive and are publicly available¹⁰. Further improvements were made over certain aspects of LSTM and GRU networks' generation process, such as time signature constraints through a bidirectional layered architecture [10], [93]. Besides recurrent neural networks, other two types have been recently used towards autonomous music generation: (i) Convolutional Neural Networks (CNNs) and (ii) Generative Adversarial Neural Networks (GANs) [95]. Succinctly, CNNs act like traditional ANNs, albeit using filters to learn recurring data patterns [96]. Even though CNNs are mainly used in image recognition, they can be adapted to work for music generation. GANs are a set of two neural networks: the generator network, whose task is to produce the wanted data and the discriminator network, that evaluates whether the output of the generator is good enough when put against ground-truth data. GANs create a feedback loop where both neural networks benefit from each other. The generator gets better at producing data closer to the real training dataset while the discriminator gets better at identifying whether the generator's output is real or artificial [97]. A well-known example of a GAN and CNN architecture for music generation is MidiNet. MidiNet is a neural network capable of generating multi-track MIDI music through symbolic representation (MIDI notes are represented by a matrix). CNNs are used in MidiNet as a generator and a discriminator. Furthermore, conditioning was applied with another CNN. This allowed for MidiNet to generate melodies conditioned on a given chord progression or a priming melody. Good results were obtained in several facets, such as hearing pleasantness and musical interest [95]. Sections 2.4.4, 2.4.5 and 2.4.6 discuss three notable systems that rely on implicit modelling. We also refer the reader to [98] for an extensive and in-depth survey on deep learning approaches for generating music.

Even though deep learning approaches have been steadily increasing in popularity as well as having a growth in quality of their generated music, these types of generative music systems also present issues at the level of control, structure, creativity, and interactivity, from a musician's perspective [99].

2.3.3 Evolutionary Approaches

Evolutionary techniques resort to the algorithms of the same name. These draw inspiration from biology. In evolutionary algorithms one has genotypes (produced genetic material during the breeding stage) which eventually turn into phenotypes (realization of genotypes). Genotypes are assessed according to fitness criteria in order to validate their transformation

¹⁰ <https://sites.google.com/view/bachprop> (Accessed March 6, 2021).

into phenotypes [34]. There has been extensive research into the development of generative music systems based on evolutionary techniques. Feature based and interactive approaches are amongst the most popular and most successful implementations of evolutionary music composition. Feature-based algorithmic music composition stems from retrieving qualitative aspects from music (e.g., if it is tonal or atonal, if there is a rhythm pattern that is strongly present in a given composition). These are then used as the fitness criteria for the evolutionary algorithms to generate music that has a strong presence of said features. Examples of this approach are in the following works:[100]–[102]. Interactive approaches work similarly to feature-based ones, with the exception of the involvement of a human component. The latter serves as the evaluator of the compositions generated by the evolutionary algorithm. In other word, the fitness criteria are solely on the human side. This approach has, naturally, advantages and disadvantages. The main advantage is that the fitness assessment is performed by the human user and as such, the results will likely converge faster to the desired goal than feature-based approaches. However, this requires the human evaluator to judge a high number of compositions generated by evolutionary methods. At a certain point, this becomes infeasible due to *user fatigue*. Furthermore, the latter may even cause deviations to occur from the desired result [10] Examples of interactive-based systems include [103]–[105].

2.4 Notable Examples

This section presents notable examples that were found when surveying the state of the art on autonomous and assisted music composition systems. Section 2.4.1 discusses Melomics, an autonomous system based on L-Systems and evolutionary algorithms. Section 2.4.2 discusses Continuator, an interactive autonomous system based on Markov Models that can play along with a human musician by responding to their input. Section 2.4.3 discusses Manousakis’ L-System implementation of an interactive autonomous system capable of generating electronic music. Section 2.4.4 discusses DeepBach, an autonomous composition tool that can generate pieces in the style of Bach Chorales using Deep Learning approaches. Section 2.4.5 discusses FlowMachines, a project by Sony CSL which aims to develop an assisted composition system using Machine Learning. Finally, section 2.4.6 discusses Dadabots, a research project consisting of neural networks that synthesize raw audio samples to generate music in a given style.

2.4.1 Melomics

Melomics is an autonomous music composition system, based on L-Systems and evolutionary algorithms. Developed by the GEB (*Grupo de Estudios en Biomimética*) of the university of Málaga, its results were positive as far as being played by a live orchestra [106]. A summarized description of the system’s approach follows. The system’s user starts by giving the system a set of parameters that represent the desired musical style of the composition that will be generated. This input data serves as the basis of stochastic processes that will generate L-

System grammars. These grammars will then produce strings that represent musical information, which are adjusted several times based on fitness functions. The final output can then be exported to different formats, such as MIDI or sheet music [34] [p. 13-65]. Melomics was further developed to be applied in the fields of therapy music [107]. The authors of this project also point to an interesting field in adaptive music composition, which that of personal music. That is, the autonomous composition system writes music, in real-time, based on the listener's surroundings, moods or other factors [34] [p.132].

2.4.2 Continuator

The Continuator was developed by François Pachet as part of a research project in the Sony Computer Science Laboratory (SCSL). Continuator is an interactive autonomous musical system, which is able to respond to a musician's input in the same style, by generating musical patterns through augmented Markov models. The system starts by analyzing an input stream that represents what a musician has just played. From there, Continuator builds trees of all possible musical patterns built from the initial input stream (i.e., the Markov Models are created). After, a global property analyzer maps properties such as tempo, meter and dynamics and produces an output stream, which will sound like what the musician played originally. The detailed algorithm of the Continuator can be found in [81].

The Continuator system proved to be interesting amongst professional musicians, but it also revealed to be successful when applied to the context of musical education with children to teach them how to develop musical behaviors at an early age [108]–[110].

2.4.3 Manousakis' Musical L-Systems

Manousakis developed a music composition tool in Max/MSP/Jitter, based on L-Systems to generate data for electronic music. In it, Manousakis mapped properties of audio waveforms to L-Systems, which allowed for a granular control of the audio that would be generated. Furthermore, Manousakis proposed a hierarchy of L-Systems, in which a main Lindenmayer System would control the behavior of its children systems. The implemented system relies on a music turtle (similar to the graphics turtle) which handles multiple properties that will generate audio. Manousakis also integrated genetic algorithms with L-Systems, to allow for a more adaptive approach to music generation. Results from this work are available online¹¹. For an in-depth explanation of Manousakis' work, we refer the reader to [15][p.49 -108].

¹¹ <https://soundcloud.com/modularbrains/do-digital-monkeys-inhabit-virtual-trees>
<https://soundcloud.com/modularbrains/undercover-hapsichord-agents-terrorize-the-court>

2.4.4 DeepBach

DeepBach is a dependency network capable of autonomously generating four-part musical pieces in the style of Bach's Chorales. Data abstraction in DeepBach was performed using symbolic representation for MIDI pitches, rhythm, time signature and other properties that are typically found within a musical piece. DeepBach is made up of a total of four neural networks, two of them being recurrent and the other two being non-recurrent. The RNNs are used to predict what notes should be played, while the non-recurrent neural networks have the responsibility of merging the results and outputting the final generated piece. Note prediction is based on pseudo-Gibbs sampling, which is a type of Markov Chain Monte Carlo algorithm. For a detailed description of DeepBach's implementation, we refer the reader to [33].

2.4.5 FlowMachines

FlowMachines is a project from Sony CSL that is about the development of an assisted music composition tool based on Machine Learning. FlowMachines can be considered an evolution of the Continuator project, making use of some of the developed techniques based on Markov Models. FlowMachines' core is in the FlowComposer system. FlowComposer relies on constrained Markov Chains, applied with other Machine Learning algorithms to generate music based on a musician's input and their own musical style. FlowMachines has been an ongoing project for ten years, and has created a lot of positive results, from a DAW plugin to musicians releasing albums with songs composed using FlowComposer [111]–[115].

2.4.6 Dadabots

Dadabots is an artistic name for a group of musicians and researchers, Zukowski and Carr. The music that is released under this name is curated from generated music created by neural networks that both researchers worked on. It should be noted that their system has also been used as an assistive tool rather than an autonomous one [116]. The neural networks are based on the SampleRNN, which is a model to generate raw audio samples [117]. This is the distinctive factor of Dadabots when compared to other approaches that rely on neural networks. By utilizing raw audio samples, no information is lost, as is typical when attempting to represent music symbolically. A downside of this approach is the amount of data required to train the network, as SampleRNN attempts to predict one audio sample at a time (meaning that, for example, if the sample rate is 44.1 kHz, the neural network will make that same number of predictions per second¹²). Notwithstanding, the results that have been obtained by Dadabots are musically relevant and interesting [118]–[120].

¹² This alludes to a common problem with using neural networks, which is the amount of computational power that is required to create good models and accurate predictions, as well as very large amounts of data.

2.5 Summary

In this chapter an extensive overview of both assisted and autonomous music composition systems was offered. After, a brief discussion related to abstracting musical information and concepts in data structures as well as in a graphical user interface was presented. Numerous examples of adaptive and nonadaptive procedures to generate music were exposed, ranging from formal grammars and evolutionary algorithms to deep neural networks. Finally, a focus was given to some notable examples of the aforementioned procedures, providing more detail into their inner workings.

3 Business Value Analysis

Value analysis is a “structured method to increase value and support the selection of the most valuable solution” [121], p.2. By examining a multitude of factors ranging from costs, product utility, customer needs, opportunity analysis and more, we can determine how to develop a product in a way that will maximize and fulfil both the customers’ and a company’s goals. As such, this chapter offers the reader a point-of-view on generative music systems that is more business-sided, contemplating the already mentioned topics. Albeit initially appearing as a completely off-scope chapter of this document, we expect the reader to conclude easily that valuable insight can be gained from such an analysis.

The outline of this chapter follows. We first offer a vision of a product that could be developed from the work that is being done on assisted music composition systems. After, we discuss innovation processes and choose the most adequate to our problem at hand. Followed by that, we discuss opportunity analysis, customer value, perceived value, benefits, and sacrifices surrounding assisted music composition systems. To close this chapter, we present Osterwalder’s value proposition model applied to our theme.

3.1 Assisted Music Composition Systems as a product

There has not been a lot of documented efforts towards commercializing generative music systems, particularly those of an assistive nature. From the research that was done, only one example was found [111]. Most work is being done on the research frontier. On the other side of the spectrum (i.e., autonomous music composition), some examples stand out [122]. Nevertheless, it is not difficult to envision an assisted music composition system as a product for the modern-day musician and composer. Digital Audio Workstations (DAWs) are computer programs used for music production (i.e., composition), recording and editing. DAWs are quite popular today, acting as the main creative tool for the modern composer [39]. Because of this, we suggest an assisted music composition system be developed as a Virtual Studio Technology instrument (VSTi) audio plugin. VSTis are programs that can run standalone or are hosted on top of a DAW which emulate a musical instrument or offer audio processing features (e.g.,

reverb or chorus effects). Moreover, they have become the standard for developing audio plugins for DAWs over the years [123]. As such, what we propose is the AMCS (acronym for Assisted Music Composition System) VSTi plugin. This plugin will offer a GUI (Graphical User Interface) that allows its users to parametrize various inputs that represent musical concepts (e.g., key, time signature, musical instrument) in order for them to get musical fragments which they can use in their own ideas in order to augment their creative process. The customer base for a product of this nature is easily identified: (a) music artists that use DAWs as their main creative tool and want another tool added to their arsenal and (b) researchers of the area of generative music systems. While the first will mainly use the AMCS plugin as a creative tool for artistic purposes, the second will be able to use the plugin as a way to evaluate and compare other existing assisted music composition tools on several criteria (e.g., complexity of the generated musical fragments and system usability). Revenue can be generated by (i) selling the plugin as a one-off purchase or (ii) selling a subscription model to have access to the plugin, although the author intends the developed project to be open-sourced and available to all. This is due to most VSTis being priced, with some reaching very high values.

3.2 Innovation Process

The theme of this work proposes the development of an assisted music composition system. These types of tools have been researched extensively for many years. However, only now and in the near future will we see their potential applied to tools that musicians and composers use in their creative process. Because of this, we need to analyze our product under an innovation process.

The innovation process, as defined by [124], can be split into three components: fuzzy front end (FFE), new product development (NPD) and commercialization. We will only consider FFE, since it is the initial phase of the innovation process. To give an analysis on the FFE, we will consider the NCD model. The NCD model admits five key elements: opportunity identification, opportunity analysis, idea genesis, idea selection and concept and technology development [125].

3.2.1 Opportunity Identification & Analysis

As the AMCS plugin is novel, several opportunities are identifiable. A new experience for musicians and composers is created, as the degree of interaction between them and the AMCS plugin is higher than regular VSTi plugins, where the user has complete control over what happens. Furthermore, a system that suggests musical ideas can also be helpful to artists suffering from writer's block [126].

3.2.2 Value Creation

Value creation stems from any tangible or intangible benefits that a product, service or activity might originate. It is also necessary to acknowledge any benefits or sacrifices that might happen during the process of consuming a given product [127]. The expected value for the end users from the AMCS plugin is that it augments the interaction between man and machine, improving the composer's compositional skills along the way by having the plugin suggest ideas that might have previously not been thought of by the composer. Benefit-wise, the end users will reduce their costs since AMCS will be open-source software available to everyone. No sacrifices were determined.

3.2.3 Value Proposition

Osterwalder's value proposition aims to position products or services around the customer's values and needs. To achieve this, Osterwalder developed the Value Proposition Canvas, in which one defines (a) the customer profile, composed of gains (the customer's expectations on what benefits are to be gained), pains (the customer's less favorable experiences in achieving the goal that the product aims to simplify) and customer jobs (the tasks or problems the customer is trying to resolve) and (b) the value proposition, composed of products and services (what is concretely being offered to the user that generates gains and resolves pains), gain creators (how the product is going to create value for the customer) and pain relievers (how the product is going to fix the customer's pains) [128]. Figure 4 presents the value proposition canvas for the AMCS VSTi plugin.

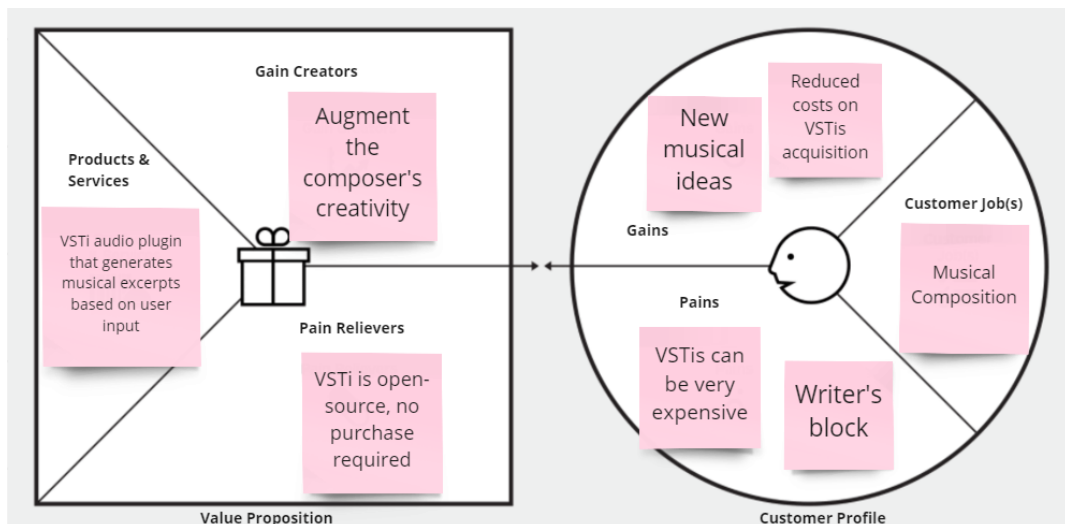


Figure 4 – Value Proposition Canvas for the AMCS VSTi plugin

For the customer profile component, the following was identified:

- Gains: new musical ideas and reduced costs on VSTis acquisition. The former is expected because the role of a music composer assumes creativity, and a continuous

search for new ideas to explore. The latter can be explained due to the AMCS VSTi being open-source and as such free of any purchases.

- Pains: high monetary costs of available VSTis and writer's block. The former can sometimes block a composer from exploring new sonic ventures simply due to a lack of monetary funds while the latter is a common psychological phenomenon that happens in a myriad of creativity related processes.
- Customer Job: in this case, only one job was identified, which is musical composition.

For the value proposition component, the following was identified:

- Products & Services: the product is a VSTi audio plugin called AMCS, which is capable of producing short musical fragments based on input that is given by a user.
- Pain Relievers: because the AMCS VSTi is open-source, musicians and composers who use it will not need to spend monetary funds to acquire it.
- Gain Creators: augmenting the composer's creativity. This is particularly important due to the role and impact that creativity has in the music writing process.

3.3 Chapter Summary

In this chapter, a business value analysis was performed over an envisioned assisted music composition system product, the AMCS VSTi audio plugin. To achieve this, the fuzzy front-end component of the innovation process was considered, which encompassed the product's value generation and its value proposition canvas.

4 Problem Analysis

This chapter provides an analysis on the problem of developing an assisted music composition system. The chapter's outline is as follows. The reader is introduced to the reasoning of choosing L-Systems for the development of the solution that is presented in chapter 6. Assumptions and limitations based on preliminary testing are presented after. A discussion about the chosen tech stack for the development of the prototype, named *L-Music*, follows. The chapter ends by capturing (i) the functional requirements of the developed prototype via Use Cases and (ii) the non-functional requirements using the FURPS+ (Functionality, Usability, Reliability, Performance, Security, Plus) model [129].

4.1 The Case for L-Systems

In chapter 2, an array of approaches was discussed for generating music in an autonomous way or an assisted way. Considering the high-quality results that each method provided, this section serves to justify why L-Systems were chosen for the development of an assisted music composition system prototype, named *L-Music*. The deciding factors were the following:

- Comparatively, the amount of research done with solely relying on L-Systems to generate music is lesser than that of other approaches (e.g., deep learning or L-Systems mixed with genetic algorithms). This motivates to further the research on L-Systems to more accurately determine where it stands with regards to the quality of the generated music when put up against other alternatives.
- Out of the reviewed approaches, L-Systems are one of the simplest in terms of its data structure and complexity of how it works. However, they can generate complex and intricate outputs. This also serves as motivation to study if the complexity of an approach influences the final outcome (e.g., is it possible to generate a melody as interesting as one that is generated by a neural network with thousands of parameters?).
- Graphical interpretations of L-Systems often result in fractal drawings or even drawings resembling plants, which also have fractal properties. Much like the latter,

fractal patterns can also be found in music [130], [131] (in fact, music can be solely generated from fractals [132]). Because the visual representations of L-Systems are visually appealing, this also motivates to further research the relationships between music, L-Systems, and fractals, i.e., if L-Systems are visually appealing and interesting, will they also be the same in an aural form?

- The author's own personal interest in L-Systems, and his curiosity on their capabilities for algorithmic music, served as the original motivation for selecting the approach to study assisted music composition.

4.2 Preliminary Testing, assumptions, and limitations

In order to validate the selection of L-Systems further, some preliminary testing was made. This testing took the form of a quick prototype¹³ implemented using the Flutter framework (the reader can refer to section 4.3.2 for an explanation of Flutter), in which the user could visualize and hear the graphical and aural representations of a given L-System respectively. The musical interpretation was implemented using an algorithm which is presented below in Figure 5.

Algorithm 1 Musical Interpretation of an L-System

Require: *sentence*, *scalePitches*, *noteSymbol*, *notes*

```

for i ← 0 to sentence.length do
  currentSymbol ← sentence[i]
  duration ← 1/8
  pitch ← 60
  note
  for j ← 1 to sentence.length do
    nextSymbol ← sentence[j]
    if currentSymbol = + then
      pitch ← scalePitches.nextPitch(pitch)
    else if currentSymbol = - then
      pitch ← scalePitches.previousPitch(pitch)
    else if currentSymbol = nextSymbol & currentSymbol = noteSymbol then duration += duration
    else if currentSymbol != nextSymbol & currentSymbol = noteSymbol then
      note ← (duration, pitch)
      notes.add(note)
      duration ← 1/8
    end if
  end for
end for
end for

```

Figure 5 - Algorithm for a musical interpretation on an L-system

An explanation follows. This algorithm assumes that a note is represented by a MIDI pitch (0-127) and a duration, in seconds. Default values are assumed for each, which are the middle C, or C4, and an eighth of a second, or an eighth note respectively. It also assumes that the "+" and "-" symbols are reserved to indicate pitch shifts. *noteSymbol* represents the symbols

¹³ The code is open-source and can be accessed at https://github.com/gildurao/flutter_turtle/tree/lindenmayer-systems

found in *sentence* which will trigger the creation of notes. Two iterations are performed over a given L-System's sentence (i.e., final output of its production rules over a given number of iterations). This is done to achieve comparison between all sentence symbols. For each symbol, the following actions may happen:

1. The pitch of the note being created can be shifted to the next or previous value found in *scalePitches*. This happens if *currentSymbol* matches the "+" or "-" symbols respectively.
2. The duration of the note being created increases by its value. This happens if *currentSymbol* and *nextSymbol* match and *currentSymbol* matches *noteSymbol*. This means that in the sentence we are seeing a sequence of *noteSymbol*. This means that the current note is going to keep being played by the L-System's sentence.
3. A note is created and added to the *notes* array. This happens when *currentSymbol* and *nextSymbol* do not match and *currentSymbol* matches *noteSymbol*. In other words, it means that a new note is to be created by the L-System's sentence, and as such, the current one needs to be saved. The duration of the next note is reset back to the default value of an eighth note.

This musical interpretation was tested with two L-Systems, representative of two well-known fractal patterns, the Koch Curve, and the Dragon Curve¹⁴. Both, while not yielding outstanding results, revealed musical patterns that could potentially be extracted by a musician to create a piece of music. The listener can also notice that those patterns repeat themselves throughout the length of the generated audio for each L-System, which points towards their fractal nature.

Besides further confirming that L-Systems are indeed apt for an assisted music composition tool, this preliminary testing allowed to identify some assumptions needed for the development of the *L-Music* prototype:

- Some symbols should be reserved for parsing a given L-System's result. Examples of this are the "+" and "-" symbols.
- The *L-Music* prototype should only concern itself with the musical interpretation of L-Systems and should allow its users to input the quality and texture of the sound they want to hear.
- The *L-Music* prototype should allow its users to input musical parameters that will enrich the final result such as rhythm, tempo, pitch range and others.

Due to their nonadaptive nature, L-Systems also impose some limitations on the *L-Music* prototype's capabilities. We must also consider the following:

¹⁴ Dragon Curve <https://soundcloud.com/user-62526899/small-dragon-curve>
Koch Curve <https://soundcloud.com/user-62526899/small-koch-curve-in-c-major>

- Musical interpretations of L-Systems will rely upon the implementer's musical knowledge. This means that the quality of *L-Music's* output will be inherently dependent on it.
- Since L-Systems are nonadaptive, some musical interpretations might not be suitable for a given musical instrument. This can be potentially solved by creating L-Systems that are tied to specific instruments or by mapping musical interpretations of an L-System to certain instruments.

Knowing the assumptions and limitations of L-Systems is necessary to select an appropriate tech stack to develop the *L-Music prototype*.

4.3 Tech Stack

As is the case for any software project, knowing which tools to use have an impact on the design and implementation of the system at hand. *L-Music* is a modular desktop application, split into three components: (i) the user interface module; (ii) the L-System module and (iii) the music generator module. *L-Music* also depends on SoundFont files to generate the audio for its users to hear the musical interpretations of L-Systems.

4.3.1 SoundFont File Format

The SoundFont (SF, or sf) file format represents a collection of synthesized audio samples that can be played as MIDI files. One can think of a SoundFont file as a digital synthesizer, which houses a number of banks, which in turn house a number of presets or instruments which contain the initially referred samples [133].

4.3.2 Flutter Framework

The Flutter framework was chosen as the toolkit to develop the user interface module, which is responsible for handling any user interactions. Flutter is an open-source framework built with the Dart programming language by Google. Flutter allows for rapid development and prototyping of user interfaces, and it targets a wide number of platforms including Android, iOS, Web, macOS, Windows, Linux, Fuchsia OS and UWP (Universal Windows Platform). It has been gaining traction exponentially due to its low learning curve, interoperability, high development speed and community [134].

4.3.3 *musicpy* and *sf2_loader* Python libraries

The L-System module and the music generator module were developed using Python. While the L-System module is responsible for the implementation of various types of L-Systems and their generation algorithms, the music generator module will take the output of the first module and interpret it in a musical perspective. The main reason for this was due to the existence of the open-source *musicpy* and *sf2_loader* libraries, which encapsulated the complexity of handling SoundFont files as well as the abstraction of musical information. The *musicpy* library allows writing music as code via a human readable syntax based on music theory concepts. The *sf2_loader* library allows control over SoundFont Files (e.g., loading soundfonts, playing audio or rendering it). It depends on *musicpy* for its audio related functionalities [113], [114].

4.3.4 Alternative Frameworks

One of the disadvantages of using Flutter and Python is that the application cannot be built as a VSTi plugin. Typically, composers and musicians who utilize software have their creative work environment setup inside a DAW application which can host multiple VSTi plugins. Because of this, other frameworks that are capable of achieving the latter were considered, namely, the iPlug2 and JUCE frameworks.

The iPlug2 framework is aimed at developing audio plugins using C++. It supports cross-platform development, targeting the Windows, macOS, Linux, Android, iOS, and Web platforms. For audio plug-in formats, it targets the VST2, VST3, AUv2, AUv3 and AAX formats. Although it supplies graphics backends to implement a user interface, it is also modular enough to allow a developer to integrate their own. It also features MIDI functionalities and DSP (Digital Signal Processing) APIs (Application Programmable Interfaces) to manipulate and render audio [137].

The JUCE framework is also targeted for audio plug-in development using C++. It targets multiple platforms: Windows, macOS, Linux, iOS, and Android. It supports the same audio plug-in formats as iPlug and also offers the same suite of features [138].

The reason both of these were rejected was because of the low-level manipulation of MIDI events and DSP they require to build a functional VSTi plugin, which was considered out of scope given the goals stated in section 1.4.. While it is a downside that the chosen tech stack does not allow to build VSTi plugins, it can run as a standalone desktop application which can be used alongside a DAW.

4.4 Requirements Engineering

This section documents the functional and non-functional requirements of the *L-Music* application. Functional requirements are captured in the form of Use Cases (UC). Non-functional requirements are captured through the FURPS+ model.

4.4.1 Functional Requirements

Functional requirements relate to what a user can achieve when interacting with a given system. *L-Music* is targeted at composers and musicians who use DAWs or notation software to write their music. Because *L-Music* is an assistive tool, it will need to request most inputs to the user (e.g., what instrument to use, or the desired musical generation type). Considering the latter, the functional requirements for the *L-Music* application are presented in Table 1.

Table 1- Functional Requirements of the *L-Music* application

| Use Case | Description |
|----------|--|
| UC01 | Load SoundFont File |
| UC02 | Select SF Instrument |
| UC03 | Play SF Instrument Sample |
| UC04 | Select L-System and number of iterations |
| UC05 | See L-System Properties |
| UC06 | Select Musical Parser |
| UC07 | See Musical Parser Description |
| UC08 | Edit Musical Parser Parameters (Scale, Rhythm, Key, Root Pitch, Beats Per Minute (BPM), Number of Octaves) |
| UC09 | Generate music given an L-System, Musical Parser and Parser Parameters |
| UC10 | Save generated music as a MIDI or a Waveform Audio File (WAV) |
| UC11 | Play, pause, and skip generated music samples |
| UC12 | Change generated music volume and playback speed |
| UC13 | Create Custom L-System |

A brief explanation of each functionality and its value to the user follows:

- **UC01:** the user should be able to load a SF file onto *L-Music*. This is to compensate the lack of integration of *L-Music* with DAW applications. By loading a SF file, the user will be able to see which instruments they want to use for their music writing goals.
- **UC02:** given UC01, the user should be able to see the list of instruments that the loaded SF file provides, as well as selecting them.

- **UC03:** the user should be able to hear what their selected instruments sound like before continuing the process of generating excerpts in *L-Music*. Each instrument, even if synthesized by audio samples, offers a different timbre and texture quality which will influence the interpretation of a composer on what the generated excerpt from *L-Music* should be used for in their pieces.
- **UC04:** because *L-Music* relies on Lindenmayer Systems for music generation, the user should, be able to select one from a predefined list as to obtain different outputs during their creative process. Additionally, due to the generative nature of L-Systems (which are further detailed in chapter 5), the user should be able to select the number of iterations for the selected L-System's generation process.
- **UC05:** given UC04, the user should be able to see a given L-System's properties, namely its axiom and rule set. This is to offer the user a chance to learn more about *L-Music* and how it works with L-Systems.
- **UC06:** the user should be able to select what type of musical interpretation to use for their selected L-System in order to hear different types of sequences or patterns. The envisioned parsers are the following: (i) a random parser; (ii) a scale-based parser; (iii) a chord progression parser; (iv) a polyrhythmic parser; (v) a polyphonic parser; (vi) a classical music parser and (vi) a jazz parser. These are further detailed in chapter 7.
- **UC07:** given UC06, the user should be offered a brief description on the behavior that the selected musical parser will have, in order to know beforehand if it is the desired parser.
- **UC08:** given UC06, and the fact that writing music is a very personal and subjective process, *L-Music* should allow the user to edit certain parameters per musical parser. Given the large number of parameters that one could edit for this, it was decided to limit the possible parameters to six: scale (e.g., C Melodic Minor scale), rhythm (which rhythmic figures are allowed to be used), key (e.g., Eb Minor), root pitch (e.g., C2), number of octaves (e.g., if the root pitch is C2 and the number of octaves is 2, the allowed range of notes goes from C2 up to and including C4) and beats per minute (BPM) (e.g., if a slower piece is desired, a value of 40 BPM might be advisable, while for a faster piece, values of 120 BPM or higher could be required). The selection of these parameters was based on the impact that each creates when writing a piece of music. With regards to this choice, in chapter 8, the reader can find the results from testing *L-Music* with composers and musicians.
- **UC09:** *L-Music* should allow the user to generate an excerpt of music given their previous inputs.
- **UC10:** the user should be able to export their generated excerpts as WAV or MIDI files. Both formats are widely used when editing audio for music purposes. Much like UC01, this also serves as compensation for the lack of DAW integration with *L-Music*. By allowing exporting the excerpts, the user can easily import them into their audio editing software of choice for further refinement (e.g., adding effects like reverb or chorus, or changing a specific sequence of notes).

- **UC11:** given UC09, the user should be able to hear the generated audio within *L-Music*. Furthermore, the application should also store all generated excerpts in case the user desires to save multiple generated music fragments.
- **UC12:** given UC11, the user should be able to change the volume and playback speed of the fragment that they are listening to. These are typical features included in DAW applications. Furthermore, changing the playback speed (i.e., changing its velocity), can prospect new ideas into the user for their music.
- **UC13:** given the reasoning presented in UC08, *L-Music* should allow its users to create their own L-Systems. This gives full control of all parameters involved in music generation within the system, which in turn will aid in increasing the user's creativity during the writing process.

4.4.2 Non-Functional Requirements

The FURPS+ model allows the definition of system-wide requirements as well as the system's qualitative attributes in several fronts (e.g., usability and reliability).

4.4.2.1 Functionality

Functionality-wise, two fields were identified, which relates to the system's capability of error handling and the system's portability (i.e., range of supported platforms).

Error Handling

- The system should be fail-tolerant in the face of user input errors.

Portability

- *L-Music* should be cross-platform and target, at least, the Windows and macOS platforms.

4.4.2.2 Usability

Usability-wise, the following fields were identified: (i) aesthetics, which encompasses all aspects related to UI (User Interface) design; (ii) learnability, which relates to how difficult it is for a user to learn to use the system effectively over time; (iii) memorability, which states that casual users of the system should be able to return to the system after extended periods of time of not using it; (iv) errors, which associates with the system's error rate as well as its gracefulness in handling them (it is influenced by the error handling requirement listed in section 4.4.2.1); (v), satisfaction, which links directly with the user's subjective opinion of the system and (vi) efficiency, which relates to how performant the system is [139].

Aesthetics

- *L-Music's* UI should be consistent in its looks and feel, meaning, for example, that different functionalities that can share equal visual components should do so to ensure a consistent and similar experience for the user.

Learnability

- Users should be able to begin generating music fragments as fast as possible using *L-Music*. This will convey that the system has a low learning curve. This is an important point, as music composition and notation software tend to have high learning curves due to their functionality set¹⁵.

Memorability

- Users should be able to understand how most, if not all, tasks are performed within *L-Music* on their first try.

Errors

- *L-Music* should inform users of their errors in an informative manner, to allow them to recover from any created issues. Additionally, the system should also have a low error rate.

Satisfaction

- The user should like to use *L-Music* and get a creative aid from it in their writing process.

Efficiency

- *L-Music* should be performant to ensure a seamless experience for the user.

4.4.2.3 Reliability

No fields were identified for reliability requirements.

4.4.2.4 Performance

Performance-wise, the efficiency field, which relates to the application's efficiency, was identified.

Efficiency

- L-System generation can be computationally expensive with a high enough number of iterations. Due to the latter, its generation algorithms should resort to concepts like recursion, whenever possible, to optimize resource consumption.

4.4.2.5 Supportability

No fields were identified for supportability requirements.

¹⁵ This is the author's opinion, based on personal experience of using such software as well as recollections from discussions in technical forums by other music composers, musicians and producers.

4.4.2.6 Plus

Plus-wise, only implementation requirements were defined.

Implementation

- *L-Music* depends on the SoundFont, MIDI and WAV file formats to function as expected.
- *L-Music*'s development should resort to open-source code whenever possible.

4.5 Chapter Summary

In this chapter, an analysis on the problem of developing an assisted music composition tool resorting to L-Systems was performed. From some preliminary testing, which encompassed the development of a quick prototype, assumptions and limitations that influenced the design and development of *L-Music* were extracted. Furthermore, it laid the ground for defining the functional and non-functional requirements of the developed system.

5 Lindenmayer Systems

Lindenmayer Systems, commonly abbreviated as L-Systems, were introduced in section 2.2.1. This chapter aims at providing an in-depth review of this formal grammar. First, the formal definition is presented, to showcase the basic structure of a Lindenmayer System. The following section, 5.2, reviews several types of L-Systems: (i) context-free L-Systems, in section 5.2.1; (ii) context-sensitive L-Systems, in section 5.2.2; (iii) stochastic L-Systems, in section 5.2.3; (iv) deterministic L-Systems, in section 5.2.4; (v) propagative and non-propagative L-Systems in sections 5.2.5 and 5.2.6 respectively; (vi) table L-Systems, in section 5.2.7 and (vii) parametric L-Systems, in section 5.2.8. [15]. After, an overview on L-System extensions is presented (e.g., having an L-System react to environment conditions) and a brief showcase of the importance of using bracket symbols with L-Systems. Finally, the selected L-System types for the development of *L-Music* are conferred.

5.1 Formal Definition

A Lindenmayer System is a generative formal grammar that relies on parallel string rewriting [40].

A formal grammar consists of three cores. The first core is an alphabet, with a finite set of terminal symbols and a finite set of non-terminal symbols that are used to form words and sentences of a formal language. These words are created according to a finite set of production rules, which is the second core. A production rule takes the form *predecessor* \rightarrow *successor*, where *predecessor* and *successor* are words made up of the referred symbols. Production rules are applied to the formal language's words with, for example, the following logic: if the word contains *predecessor*, replace it with *successor*. The last core of a formal grammar is a starting symbol, from which the remainder of the formal language is defined. This is commonly referred to as the axiom. Therefore, we can formally define generative grammars with Chomsky's proposal [140].

Let G be a formal generative grammar, represented by the set $G = \{N, S, \omega, P\}$, where:

- N is a finite set of replaceable, non-terminal symbols.
- S is a finite set of terminal symbols, disjoint from N .
- ω is a string of symbols from N and defines the initial state of a system, i.e., the axiom.
- P is a finite set of production rules, which take the form *predecessor* \rightarrow *successor*, and are applied iteratively starting from ω .

As stated at the beginning of this section, L-Systems are also classified as a generative formal grammar, taking the form of a triplet G , where $G = \{V, \omega, P\}$, being that V is the system's alphabet, ω the system's axiom and P the system's production rules (for a more complete formal definition, the reader should refer to section 2.2.1). Where they diverge from a standard Chomsky grammar is that their string rewriting mechanism is parallel, whereas in a Chomsky grammar, string rewriting is sequential.

The next section contemplates multiple types of L-Systems. These types are named according to the nature of the L-System's grammar, i.e., how production rules are applied to the string rewriting mechanism.

5.2 Types

This section introduces several types of L-Systems. In order, they are:

- Context-free L-Systems.
- Context-sensitive L-Systems.
- Stochastic, or non-deterministic, L-Systems.
- Deterministic L-Systems.
- Propagative L-Systems.
- Non-propagative L-Systems.
- Table L-Systems.
- Parametric L-Systems.

It should be noted that a given L-System can possess several types if desired. We also offer an overview on applying extensions to L-Systems, specifically having them be sensible to environment conditions, and establishing hierarchy between a set of L-Systems. Additionally, we go over the importance of adding bracket symbols to an L-System's alphabet and their standard interpretation of creating a state for the interpretation of the system's output. We end the section by providing the selected L-System types for the development of this project.

5.2.1 Context-free (OL)

Context-free L-Systems can be considered the simplest. Its production rules take the standard form of *predecessor* \rightarrow *successor*, where *predecessor* is a symbol of the system's alphabet and *successor* is a word of the alphabet's language.

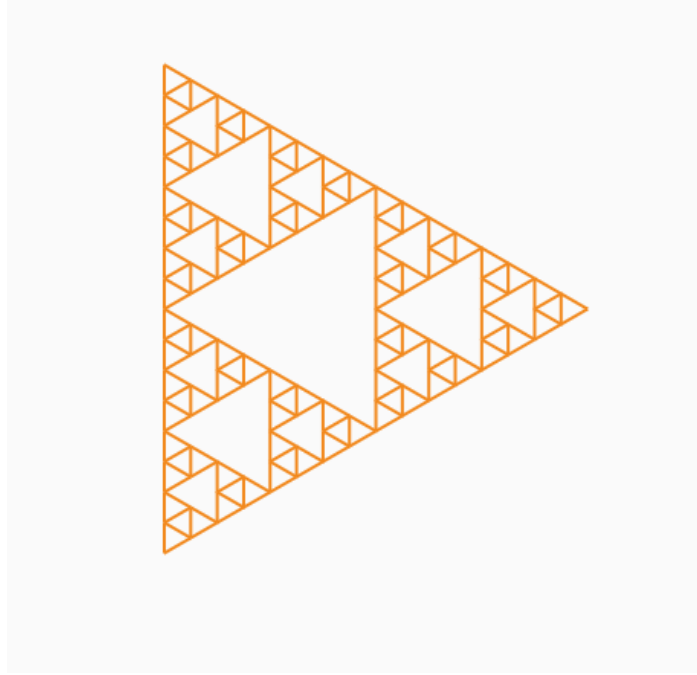


Figure 6 – Sierpinski Triangle generated from an L-System

5.2.2 Context-sensitive (IL)

Context-sensitive L-Systems' production rules [141] take the form of:

left context < *predecessor* > *right context* \rightarrow *successor* , meaning that for the production rule to apply, the *predecessor* needs to be surrounded by both contexts. Either *left context* or *right context* can be empty (usually written as the symbol \emptyset), meaning that the condition for that context is always validated. A production rule with an empty context can be referred to as a 1L rule, whereas a rule with two contexts can be referred to as a 2L rule. An example follows:

Let L be a context-sensitive L-System grammar, such that:

Its alphabet is $V = \{a, b, c, d\}$.

Its axiom is $\omega = "abab"$.

Its production rules are

$P = \{ "ab < a > \emptyset \rightarrow c", " \emptyset < b > ab \rightarrow d", " \emptyset < c > \emptyset \rightarrow abab" \}$.

This means that:

- For predecessor "a" to turn into successor "c", it needs to encounter the word "ab" on its left (context). Nothing needs to be validated for its right context, as it is empty.

- For predecessor “b” to turn into successor “d”, it needs to encounter the word “ab” on its right (context). Nothing needs to be validated for its left context, as it is empty.
- Predecessor “c” will turn into word “abab” whenever it is encountered, since both of its contexts are empty.

Considering n to be the number of iterations over which we apply the production rules on ω , the following outputs can be obtained:

$$n = 0 \rightarrow abab$$

$$n = 1 \rightarrow adcb$$

$$n = 2 \rightarrow adababb$$

$$n = 3 \rightarrow adadcbbb$$

$$n = 4 \rightarrow adadababbbb$$

Context-sensitive grammars are useful to impose conditions on the application of a system’s production rules. For example, one could use context-sensitive rules to restrict the selection of what musical notes or chords can be played in the absence or presence of other notes or chords.

5.2.3 Stochastic (non-deterministic)

Stochastic L-Systems are non-deterministic, meaning that there is a probability distribution associated with the grammar’s production rules. Stochastic L-Systems can be defined as a triplet $G(\pi)$, where $G(\pi) = \{V, \omega, P\}$: V is the system’s alphabet, ω is the system’s axiom, P is the set of production rules. π represents the probability distribution function over P , which maps each production rule to a probability. For the same predecessor in P , the sum of probabilities must be equal to 1 [142]. Stochastic production rules take the form of:

$$\text{predecessor} \xrightarrow{\text{probability \%}} \text{sucessor}$$

Let L be a Stochastic L-System grammar, such that:

Its alphabet is $V = \{a, b\}$.

Its axiom is $\omega = "a"$.

Its production rules are $P = \{ "b \rightarrow a", "a \xrightarrow{70\%} ab", "a \xrightarrow{30\%} ba" \}$.

This means that there is a 70% chance of predecessor “a” being rewritten as successor “ab” and a 30% change of it being rewritten as successor “ba”.

Consider n to be the number of iterations over which we apply the production rules on ω , the following outputs can be obtained over 5 iterations:

$$n = 0 \rightarrow a$$

$$n = 1 \rightarrow ab$$

$$n = 2 \rightarrow baa$$

$$n = 3 \rightarrow abaab$$

$$n = 4 \rightarrow baaababa$$

or

$$n = 0 \rightarrow a$$

$$n = 1 \rightarrow ab$$

$$n = 2 \rightarrow aba$$

$$n = 3 \rightarrow ababa$$

$$n = 4 \rightarrow abaababa$$

5.2.4 Deterministic (DL)

Deterministic L-Systems are all L-Systems for which it is possible to determine the exact output for any number of given iterations. In other words, every word or symbol only appears once as a predecessor in the finite set of production rules. The example given in section 5.2.1 is deterministic, in addition to being context-free.

5.2.5 Propagative (PL)

Propagative L-Systems are all L-Systems that contain at least one production rule where the *successor* is a word consisting of at least two symbols. Having production rules succeeding in longer words will amplify the effect of the L-System's growth. The example given in section 5.2.2 is propagative, in addition to being context-sensitive [143].

5.2.6 Non-Propagative

Non-propagative L-Systems are all L-Systems that only contain production rules where the *successor* is one symbol. This means that the axiom of these systems will state the length of the generated string, regardless of the number of iterations one performs over it [143].

An example of a non-propagative deterministic context-free L-System follows:

Its alphabet is $V = \{a, b\}$.

Its axiom is $\omega = "ab"$.

Its production rules are $P = \{ "a \rightarrow b", "b \rightarrow a" \}$.

Considering n to be the number of iterations over which we apply the production rules on ω , the following outputs are obtained over five iterations:

$$n = 0 \rightarrow ab$$

$$n = 1 \rightarrow ba$$

$$n = 2 \rightarrow ab$$

$$n = 3 \rightarrow ba$$

$$n = 4 \rightarrow ab$$

5.2.7 Table L-Systems

Table L-Systems are all L-Systems whose production rule set is comprised of tables of production rules. The condition to switch between tables is up to the creator of the Table L-System. They were first introduced to simulate environment changes during a plant's development [144].

Let L be a Table L-System grammar, such that:

Its alphabet is $V = \{a, b\}$.

Its axiom is $\omega = "b"$.

Its production rule tables are:

$$Table\ 1 = \{ "a \rightarrow ab", "b \rightarrow a" \}$$

$$Table\ 2 = \{ "a \rightarrow b", "b \rightarrow ba" \}$$

Considering (i) n to be the number of iterations over which we apply the production rules on ω and (ii) that Table 1 is used first and that from $n = 3$ Table 2 is used, the following outputs are obtained over five iterations:

$$n = 0 \rightarrow b (Table\ 1\ rules\ are\ used)$$

$$n = 1 \rightarrow a (Table\ 1\ rules\ are\ used)$$

$$n = 2 \rightarrow ab (Table\ 1\ rules\ are\ used)$$

$$n = 3 \rightarrow bba (Table\ 2\ rules\ are\ used)$$

$$n = 4 \rightarrow babab (Table\ 2\ rules\ are\ used)$$

5.2.8 Parametric

Parametric L-Systems are all L-Systems who are defined as a quadruplet and whose words are parametric, i.e., they are associated with a set of parameters. They were proposed out of a necessity of handling real numbers for a more accurate simulation of certain plant growth mechanisms. Let G be a parametric, context-free L-System, where $G = \{V, \Sigma, \omega, P\}$.

V is its alphabet.

Σ is its set of formal parameters.

ω is its axiom.

P is its set of production rules.

Parametric production rules take the form of:

$$predecessor : condition \rightarrow successor$$

Where $predecessor \in V \times \Sigma^*$ (formal module), $condition \in C(\Sigma)$ (logical expression) and $successor \in (V \times \varepsilon(\Sigma)^*)^*$ (formal parametric word) [145].

Let L be a parametric context-free L-System grammar, such that:

Its alphabet is $V = \{A, B, C, D\}$.

Its formal parameter set is $\Sigma = \{t\}$

Its axiom is $\omega = "B(5)A(7)"$.

Its production rules are $P = \{ "A(t) : t > 4 \rightarrow B(t-7)CD(t-1, t-2)A(t-1)" \}$.

Considering n to be the number of iterations over which we apply the production rules on ω , the following outputs are obtained over four iterations:

$$n = 0 \rightarrow B(5)A(7)$$

$$n = 1 \rightarrow B(5)B(0)CD(6,5)A(6)$$

$$n = 2 \rightarrow B(5)B(0)CD(6,5)B(-1)CD(5,4)A(5)$$

$$n = 3 \rightarrow B(5)B(0)CD(6,5)B(-1)CD(5,4)B(-2)CD(4,3)A(4)$$

Parametric L-Systems essentially create a door for L-Systems to manipulate numbers besides strings in their parallel rewriting process.

5.2.9 Applying Extensions on L-Systems

All previously mentioned L-System types can be further extended with added functionalities. Some of these examples are (a) environmentally sensitive L-Systems, which are an augmentation of parametric L-Systems to include the influence of outside environment factors (e.g., in the case of plant growth and development, one of these factors could be intensity of sunlight); (b) open L-Systems, which make the environment a parallel system to the L-System instead of an outside influence. This implies that both the L-System and the environment communicate with each other and model each other's behavior; (c) multi-set L-Systems, which attempt to simulate a population of individuals, interacting with each other rather than just a single individual (e.g., a forest versus a single tree) and (d) hierarchical L-Systems which are a network of L-Systems. A main L-System propagates production rules to child L-Systems based on its own imposed conditions [15].

5.2.10 Using brackets to manage an L-System's interpretation state

In section 2.2.1 an L-System that relied on the bracket symbols, `[]`, was presented. Its graphical representation, using turtle graphics, resembles a bush-like plant. Brackets are usually interpreted as state management symbols, which in turn allows for a more fine-grained control of how the L-System's growth will be when it is parsed graphically or in another fashion. Brackets are particularly useful for scenarios where similar branches of the L-System need to happen at different points in space (in the case of turtle graphics) or time (in the case of audio generation).

5.2.11 Chosen L-System types for the development of *L-Music*

Having a wide pool of L-System types creates a lot of space for creating interesting musical results. However, the challenge of creating those same results with simple data structures (i.e., a context-sensitive L-System is much less complex than an environmentally sensitive L-System) is also considered interesting. Furthermore, the properties of data amplification that L-Systems have allows us to have complex results with simple rules. The following types will be considered for the development of the *L-Music* application:

- Context-free and context-sensitive.
- Propagative and non-propagative.
- Deterministic and non-deterministic (stochastic).
- Table L-Systems.

5.3 Chapter Summary

This chapter served as an introduction to the topic of L-Systems by explaining to the reader how L-Systems derive from a Chomsky grammar. By exposing several L-System types, such as context-free and parametric systems, it is expected that the reader gained an overall understanding of how small modifications to how production rules work in a grammar drastically change their output. Furthermore, the importance of the bracket symbols for interpreting L-Systems were highlighted. The chapter ended with the selection of L-System types to be used in *L-Music*.

6 Design

This chapter comprises of the design process for the implementation of *L-Music*. Its outline follows. The first part of this chapter provides the system's design, represented using the 4+1 View Model [129][p. 501 – 503] for an understanding of *L-Music* as a whole. Along with the latter, the application design for *L-Music*'s three modules, the user interface module, the L-System module, and the Musical Parser module, is documented for a better understanding of each module's responsibility. The second part of this chapter documents the key components of *L-Music*'s User Interface (UI) and User Experience (UX) design, specifically, (a) the application's color scheme; (b) a brief mention of Google's Material Design and its integration with the Flutter framework; (c) concerns with regards to UX: visual hierarchy and flow, component behavior consistency and the use case related to creating a custom L-System.

6.1 System and Application Design

This section comprises of *L-Music*'s system and application design. The system design is captured using the 4+1 View Model, illustrated by (i) the logical view, which aims to expose the most important layers of the system; (ii) the deployment view, which showcases how the system will be running physically on an end user's machine; (iii) the implementation view, which is the representation of the system's implementation and (iv) the process view, which shows how a user interaction with the system is process in a generic fashion. The application design displays how each module is organized, and what software architecture and development patterns were applied to ensure good practices.

6.1.1 4+1 View Model

The 4+1 View model's purpose is to document and describe a system's architecture, giving emphasis to its structure, modularity and essential components and its reason of being designed the way it is [129][p. 501 – 503]. The four views presented here are considered the

most valuable ones due to the latter. However, there can be more, such as the Security View. Each view is embodied by Unified-Modelling-Language (UML) [129] [p. 10] diagrams.

6.1.1.1 Logical View

The logical view is represented by a System Component Diagram, visible in Figure 7.

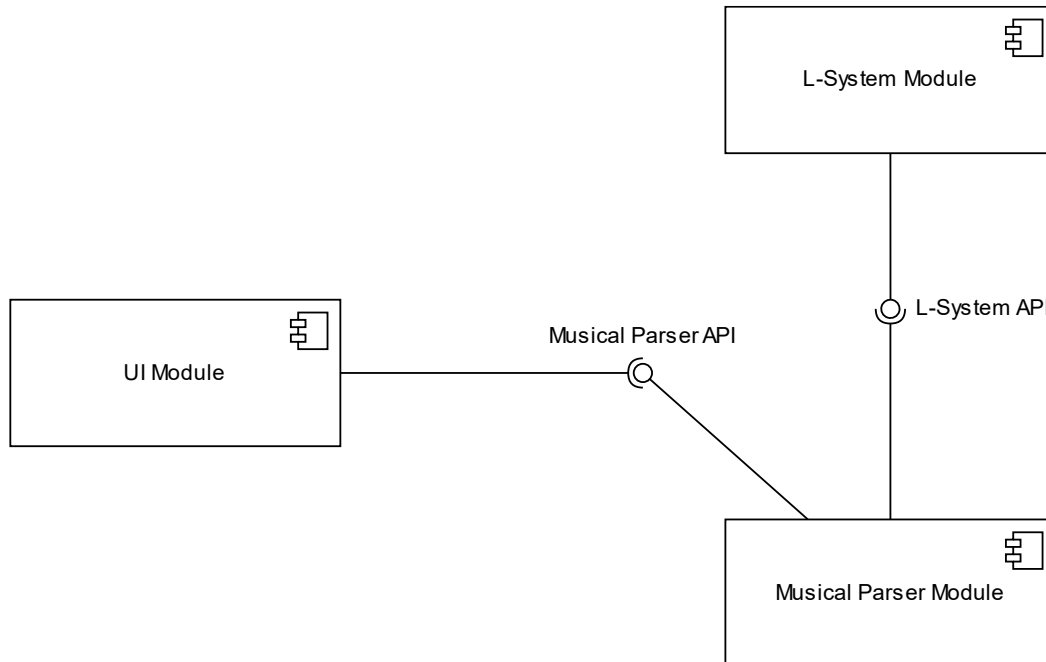


Figure 7 - *L-Music* System Component Diagram

L-Music consists of three components: the UI Module, responsible for handling all user interactions. It consumes the Musical Parser API, which is exposed by the Musical Parser Module. The latter handles any operations related to manipulating SoundFont Files as well as generating musical fragments based on user input received by UI Module. Since this generation depends on creating L-Systems, the Musical Parser Module consumes the L-System API, exposed by the L-System Module. The L-System Module is responsible for creating an operating on L-Systems, creating the required output for the Musical Parser Module to generate audio. This results in a separation of concerns between each component (i.e., a layered architecture), allowing the replacement of each to be seamless if needed be.

6.1.1.2 Deployment View

The deployment view is represented by a Deployment Diagram, visible in Figure 8.

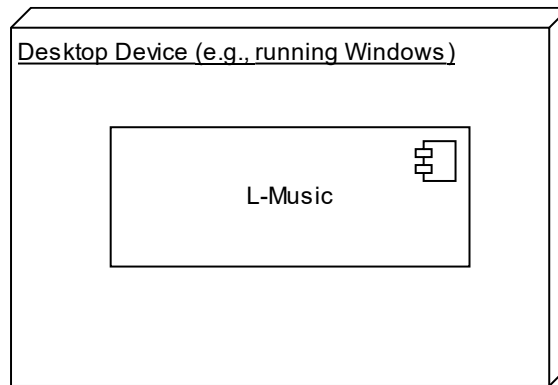


Figure 8 - *L-Music* Deployment View

The deployment view is rather simple due to *L-Music* being a standalone desktop application, i.e., it does not need to communicate and consume resources from external sources of information. Therefore, it only needs to be hosted on the end user's desktop device. This ensures that any possible latency in communicating with the user, which is a crucial aspect with applications that handle audio-based operations, is reduced.

6.1.1.3 Implementation View

The implementation view is represented by a package diagram, visible in x.

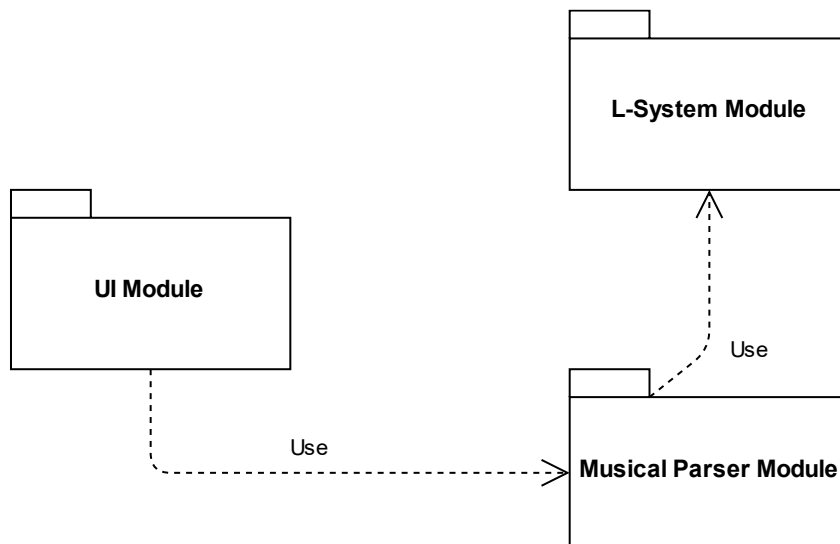


Figure 9 - *L-Music* Package Diagram

The package diagram of *L-Music* is equal to its system component diagram, meaning that the UI Module has a dependency on the L-System Module and the Musical Parser Module has a dependency on the L-System Module.

6.1.1.4 Process View

The process view is represented by a sequence diagram illustrating a generic action with the system, visible in Figure 10.

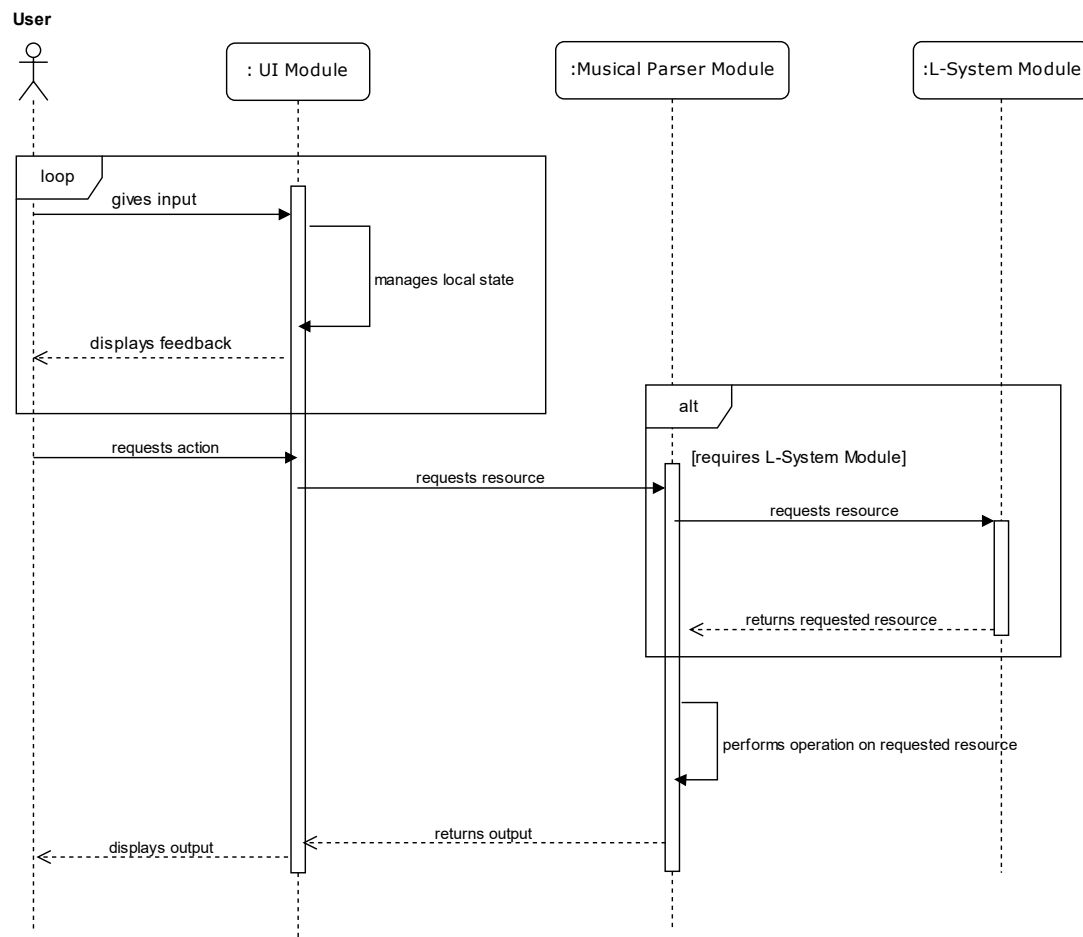


Figure 10 - *L-Music* Sequence Diagram

An explanation of the diagram in Figure 10 follows. The user starts by giving input (e.g., selecting a given L-System) to the UI Module. The UI Module will manage its local state to save the user's input over time and displays feedback accordingly. When the user is done with providing input, they may request an action to the UI Module. This action is translated into a resource request to the Musical Parser Module (e.g., hearing what an instrument sounds like or generating a new audio excerpt). If the requested resource requires the L-System Module, the Musical Parser Module will request the appropriate resource from the former. In both cases, the latter performs the necessary operations on the initially requested resource, returning the output to the UI Module, who displays it to the user.

6.1.2 Application Design

This section documents the design at an application level for each of the core modules of *L-Music*, those being the UI Module, the L-System Module, and the Musical Parser Module.

6.1.2.1 User Interface Module

The UI module is split into four layers:

- The presentation layer, responsible for graphical user interface and handling user interactions.
- The application layer, responsible for communicating between the presentation layer and the data and domain layers.
- The domain layer, responsible for organizing concepts from the problem at hand into objects.
- The data layer, responsible for making requests to the Musical Parser Module and returning its outputs to the application layer.

Figure 11 illustrates the component diagram for the UI Module.

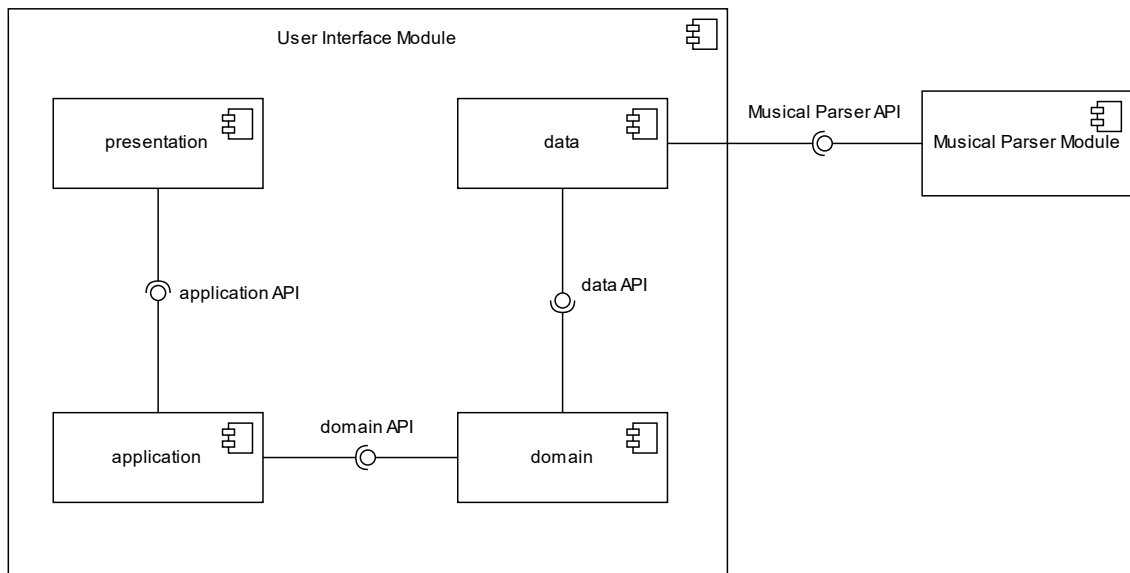


Figure 11 - User Interface Module Component Diagram

Much like the system's architecture, splitting atomic responsibilities per layer allows us to replace any of them, when necessary, without impacting the remaining layers of the UI Module.

6.1.2.2 L-System Module

The L-System Module's responsibility has to do with representing the concept of an L-System. As written in section 5.1, L-Systems consist of an alphabet of symbols, an axiom, which is a sequence of symbols from the alphabet and a set of production rules, which state how symbols in the axiom should be treated. Besides the aforementioned properties, it is also feasible to consider that an L-System has a generation function, in which one performs the

parallel string rewriting mechanism. Therefore, we can model an L-System class using Object Oriented Design concepts [129] [p.7], exemplified in Figure 12.

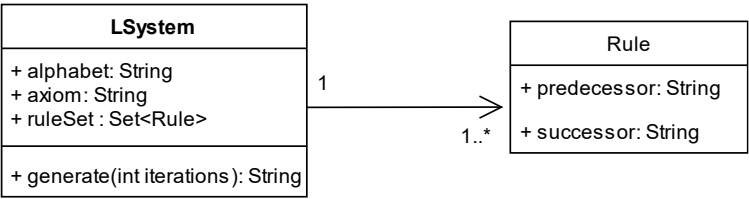


Figure 12 – Object representation of a Lindenmayer System

Further extending the presented design, the Inheritance concept was applied to handle the selected types of L-Systems. This extension is showcased in Figure 13.

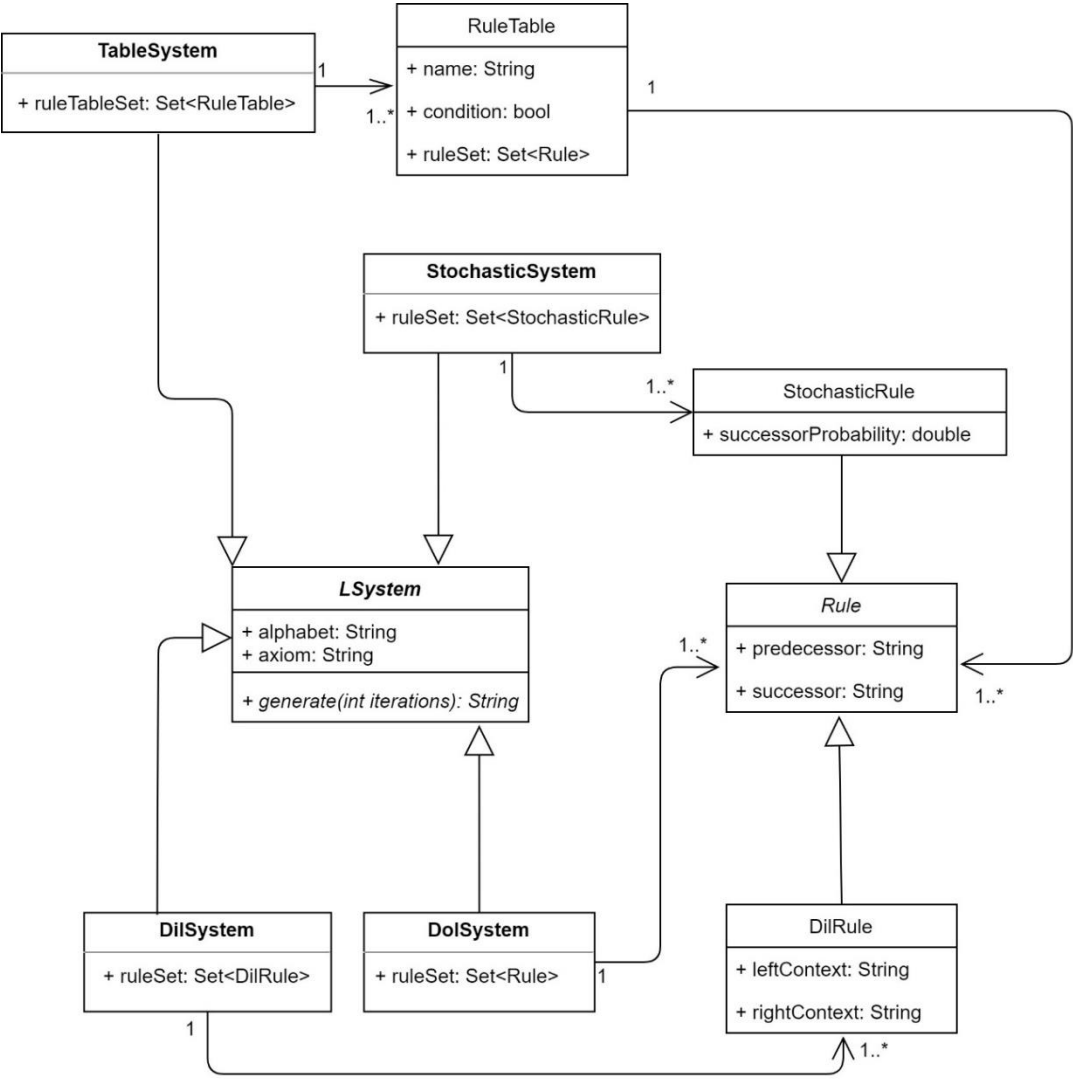


Figure 13 - L-System Module Class Diagram

An explanation follows. The *LSystem* and *Rule* classes were made abstract, to accommodate the fact that both have different types. The attribute *ruleSet* was removed from *LSystem* due

to the same reason (the reader should note that the *TableSystem* class does not possess a set of *Rule* objects, but rather a set of *RuleTable* objects which contain *Rules* themselves). The *generate* method was also made abstract due to L-Systems of different types having different means of interpreting their production rules. Concrete classes were created per selected L-System type:

- *DoSystem* refers to deterministic context-free L-Systems. Besides inheriting the attributes of *LSystem*, it possesses a set of *Rule* objects.
- *DilSystem* refers to deterministic context-sensitive L-Systems. Besides inheriting the attributes of *LSystem*, it possesses a set of *DilRule* objects.
- *StochasticSystem* refers to stochastic L-Systems. Besides inheriting the attributes of *LSystem*, it possesses a set of *StochasticRule* objects.
- *TableSystem* refers to Table L-Systems. Besides inheriting the attributes of *LSystem*, it possesses a set of *RuleTable* objects.

Propagative and non-propagative L-Systems were not considered here because these types are dependent on the nature of the predecessors and successors of an L-System's production rule set rather than the L-System itself.

The same logic is applied to represent production rules:

- *Rule* represents a context-free deterministic production rule. It possesses two String objects: *predecessor* and *successor*.
- *DilRule* represents a context-sensitive deterministic production rule. Besides inheriting the attributes of *Rule*, it possesses two String objects: *leftContext* and *rightContext*, which represent the left context and right context of a context-sensitive rule (as discussed in section 5.2.2).
- *StochasticRule* represents a stochastic production rule. Besides inheriting the attributes of *Rule*, it possesses a double value, *successorProbability*, which represents the probability of a given successor to be chosen for a given production rule (as discussed in section 5.2.3).
- *RuleTable* represents a production rule table that Table L-Systems depend on. It has the following attributes: *name*, a String, represents the table's name; *condition*, a boolean value, represents the condition for which the *RuleTable* object in question should be used and *ruleSet*, a Set of *Rules* represents the table's production rules set (as discussed in section 5.2.7).

This design allows the addition of other L-System types due to the abstraction of the elements that make up an L-System. For example, if a context-free stochastic L-System is desired, one can create a class named *StochasticLSystem* which would extend *LSystem* and have a set of *StochasticLRule* objects, which would extend *Rule* and have two String objects, *leftContext* and *rightContext* and a double value, *successorProbability*.

6.1.2.3 Musical Parser Module

The Musical Parser Module is responsible for manipulating SoundFont files and generating music fragments from the output provided by the L-System module. This module can be

considered as a list of functions that perform the referred manipulations. The list is the following:

- *loadSf2*, which loads a SoundFont file from *soundFontFilePath*.
- *saveFile*, which saves a generated piece of music onto a file, either in WAV or MIDI format.
- *changeInstrument*, which changes the instrument (i.e., SoundFont preset) that is currently selected by the user.
- *generateInstrumentSample*, which plays a small predefined musical excerpt so the user can hear what their selected instrument sounds like.
- *generateScaleSample*, which plays a scale given by the user with their selected instrument.
- *generateMusic*, which generates a music fragment based on user input.

6.2 UI/UX Design

Given that *L-Music* is a tool to *assist* musicians and composers in their creative craft, the design of the user interface and user experience should be contemplated and well thought out. This section documents all concerns that were taken regarding the latter. The first section, 6.2.1, establishes the application's color scheme and maps each defined color to a meaning. The second section, 6.2.2, offers a brief description of Google's Material Design guidelines and the reason for using it when creating the interface's components. The third section, 6.2.3, documents some concerns regarding *L-Music's* user experience, specifically: (a) visual hierarchy and flow; (b) component consistency and (c) the use case of creating a custom L-System.













6.2.1 Color Scheme

The color scheme for *L-Music* was defined according to the following criteria:

- The color scheme's colors that are used to display texts should follow the Web Content Accessibility Guidelines (WCAG) for colors [146]. The latter defines two levels of contrast ratio that should be followed for normal (at least 4.5:1), large (at least 3:1), and bold (at least 3:1) texts [147] given a background color for those texts.
- The color scheme should be defined with as few colors as possible to reduce the user's memorability of interacting with the application [148] [p.46-47].
- The color scheme's colors should be mapped to a relevant utility (e.g., relaying to the user that an item is selected) [148] [p.46]. This is also an effort to reduce user memorability and lower the learning curve of *L-Music*.

Considering the defined criteria, the color scheme for *L-Music* is presented in Table 2. The colors were obtained through a process of trial and error using a color palette generator.

Table 2 - *L-Music's* Color Scheme

| Color Example | Hexadecimal Code and Name | Utilities | Contrast Ratios per WCAG Standards |
|--|-------------------------------------|--|---|
| A:  | #0E0D0E Rich Black FOGRA 39 | Background Color | - |
| B:  | #F7D08A Deep Champagne | Actionable Texts, List view highlighter | 13.34:1 with A |
| C:  | #F487B6 Persian Pink | Indicates that a list item is selected | 8.36:1 with A |
| D:  | #5C678A Dark Blue Gray | Inactive track slider color | - |
| E:  | #FAFFFD Baby Powder | Audio related information texts | 19.33:1 with A |
| F:  | #36443E Kombu Green | Dropdown Background Color | - |
| G:  | #CAE5FF Beau Blue | Active track and thumb slider color | - |
| H:  | #CCC5B9 Pale Silver | Hint Texts | 11.32:1 with A |
| I:  | #6D98BA Cerulean Frost | Section highlighter | - |
| J:  | #F7A9A8 Pastel Pink | Error Messages, Delete actions | |
| K:  | #ECEBE4 Alabaster | Non-actionable texts and non-selected items, Scrollbar color | 16.23:1 with A 10.29:1 with L 8.56:1 with F |
| L:  | #1A3C1D Forest Green Traditional | Success message background color | - |
| M:  | #AAF8A0 Light Green | Loading Indicators | - |

A breakdown of Table 2 follows:

- Colors *A*, *F* and *L* are background colors, specifically *A* serves as the main background color of the app; *F* serves as the background color for the dropdown components of the app and *L* serves as the background color for any success messages in the app (e.g., when a file of a musical fragment is successfully saved by the user's request).
- Colors *B*, *C*, *E*, *H*, *J* and *K* are text colors. The color *B* is used for actionable texts (e.g., buttons). Color *C* is used for conveying the user that a list item is selected. Color *E* is used for audio related text information (e.g., the current volume and playback speed of the audio file). Color *H* is used for hint texts (e.g., providing the user some help to being creating their custom L-System). Color *J* is used for error messages or delete actions (e.g., removing a production rule when creating a custom L-System). Color *K* is used for non-actionable texts and non-selected items (e.g., the number of beats per minute that the generated music fragment will have).
- Colors *D* and *G* serve to indicate the inactive and active portions of a slider's track respectively. Additionally, color *G* also serves as the color for a slider's thumb (i.e., the component of a slider with which the user can control the slider's value by sliding it across the slider's track).
- Color *I* is used to highlight sections of components. In section 6.2.3.1 these sections are showcased.
- Color *M* is used for any loading indicators that may be needed to convey the user a feeling of progress and that the system is actively working on their requested action.

Colors are not the only way to convey information to the user about how they should use or interpret the system. It is also necessary to establish what components to use. For this purpose, the Material Design guidelines and components were used.

6.2.2 Material Design

Material is a design system developed by Google. It provides a standardized list of components and design guidelines to follow when developing an application. While originally developed for mobile devices, Material can also be applied to desktop and web applications (e.g., the Gmail web application is implemented using Material design) [149]. Although having a specific design system might prove beneficial towards an application such as *L-Music*, Material Design was chosen for the following reasons:

1. A majority of digital users (i.e., users who utilize computers, phones, tablets, etc.) are already familiar with a Material UI due to the outreach of Google's applications. This aids in reducing the learning curve of *L-Music*.
2. Material Design Components are already implemented in the Flutter framework, and as such, development of the *L-Music* prototype can be done at a quicker pace than if one had to implement all visual components from scratch and reiterate over them.

There are, naturally, disadvantages to constraining *L-Music* to follow a design system such as Material. One of them being the dependency that is created on the application by the design

system. Another disadvantage is that if Material proves to be an ineffective design for assisted music composition systems, *L-Music* needs to be entirely redesigned.

Given that Material already provides a standard list of components to use for building user interfaces [150], the following is a list of the ones that were chosen for *L-Music*:

- Text Buttons. This component was chosen for buttons where a textual description of the action would be more valuable to the user than just using an Icon (e.g., a button to indicate that the user can generate an audio excerpt).
- Icon Buttons. This component was chosen for buttons where imagery would suffice to convey the right message to the user (e.g., a trash icon is recognizable to most users as a delete functionality).
- Lists. This component was chosen to organize large amounts of data in a concise way (e.g., the instruments list from a SoundFont file).
- Progress Indicators. This component was chosen so the system could convey to the user that it is still working while operations that may take a long time are being performed.
- Snackbars. This component was chosen to convey to the user of success messages (e.g., that a file with the desired music fragment was saved successfully in the user's device).
- Dropdown Menus. This component was chosen to organize large amounts of data in components that had less priority in visual hierarchy than the ones that depend on lists. For example, the L-Systems list or the instruments list needs to be immediately noticed by the user, as well as its extent, whereas the list of root pitches does not need to be noticed in its full extent, and as such, can be hidden under the context of a dropdown menu.
- Sliders. This component was chosen for all audio related manipulation operations. This means volume control, audio position indication (i.e., visualizing its duration and conveying it to the user) as well as the audio's playback speed.
- Text Fields. This component was chosen due to the particular use case of creating a custom L-System. Due to its nature, meaning that it involves creating a form for the user to fill out, text fields were a necessity.

6.2.3 UX Concerns

Some UX concerns were identified for *L-Music's* user experience, namely about visual hierarchy and flow, component consistency and the use case of creating a custom L-System. Rough hand-drawn sketches of the initial UI design accompany each concern.

6.2.3.1 Visual Hierarchy and Flow

Establishing a visual hierarchy is important for an application to allow a user to understand where and how they need to start interacting with the system. Furthermore, one should also keep in mind of how much information the user is presented with at the beginning of their

experience with the system [148] [p.48-51]. Figure 14 shows a rough sketch of the first screen the user would see when opening *L-Music*.

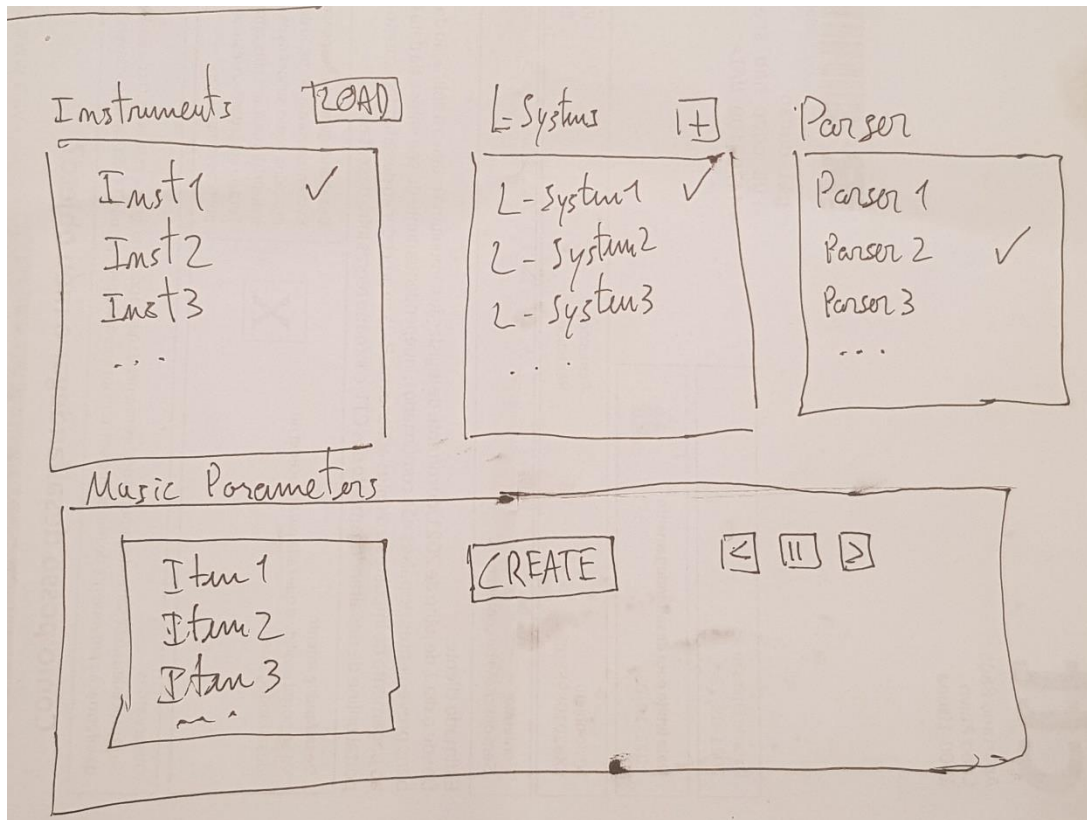


Figure 14 - Rough Sketch of *L-Music*'s initial landing screen

By analyzing Figure 14, we can extract the following:

- The user is presented with a lot of information to absorb when first opening the application. While this may not be an issue for frequent users of *L-Music*, users that would use it temporarily might find it hard to memorize and relearn.
- Due to the aforementioned large amount of visual information to take in, the user might be confused as to whether they need to look at first. Is it the music parameters section? Or is it the L-Systems list? This creates a lack of visual hierarchy and does not suggest a flow of actions to the user.

These two problems can be solved by just displaying the instruments list in an empty state and informing the user that they need to load a SoundFont file. The reiteration of the initial landing screen is visible in Figure 15.

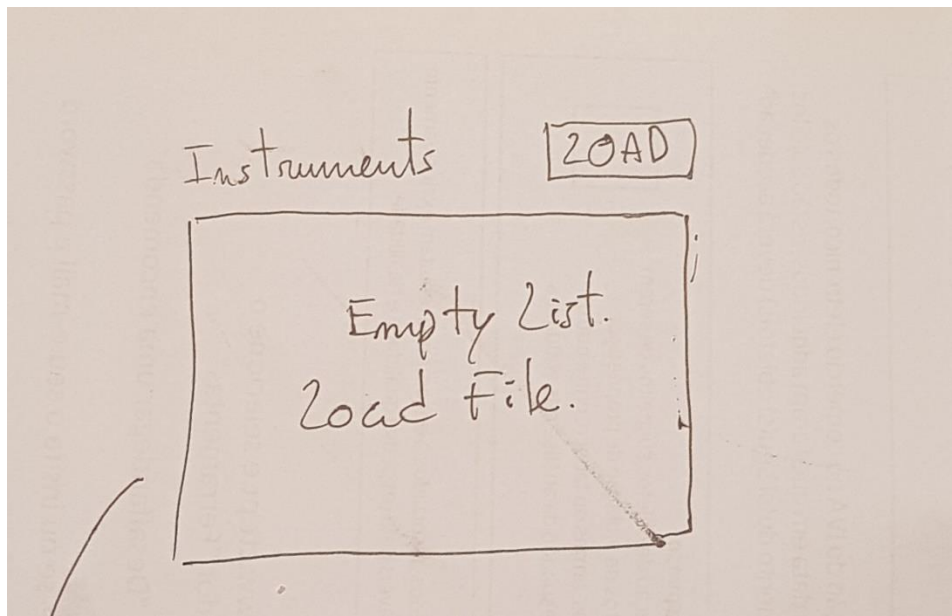


Figure 15 - Reiteration on *L-Music*'s initial landing screen design

By only showing the instruments list in an empty state, with an appropriate message for the user (i.e., indicating them that they should load a file onto the system), as displayed in Figure 15, a visual hierarchy and flow for the application's interactions is achieved. This implies that the remainder of the UI will only be shown when a SoundFont file has been loaded by the user. The user will only have one available action as an entry point to utilizing the system. This can, however, prove frustrating to more advanced users of *L-Music*. The system should accommodate all levels of experience [148] [p.52]. In section 9.2, this is listed as a topic for future work.

6.2.3.2 Component Consistency

Component consistency refers to reusing the same type of graphical user interface for different, or similar, functionalities. In other words, it is beneficial for a UI to be redundant since it will simplify the experience for the user, lowering the system's learning curve. Besides this, it will also result in less errors from the user, since the same component is being reused, albeit for different purposes (i.e., the user will know what to expect from that component if they see it in a different part of the system). Figure 16 serves as an example of component consistency.

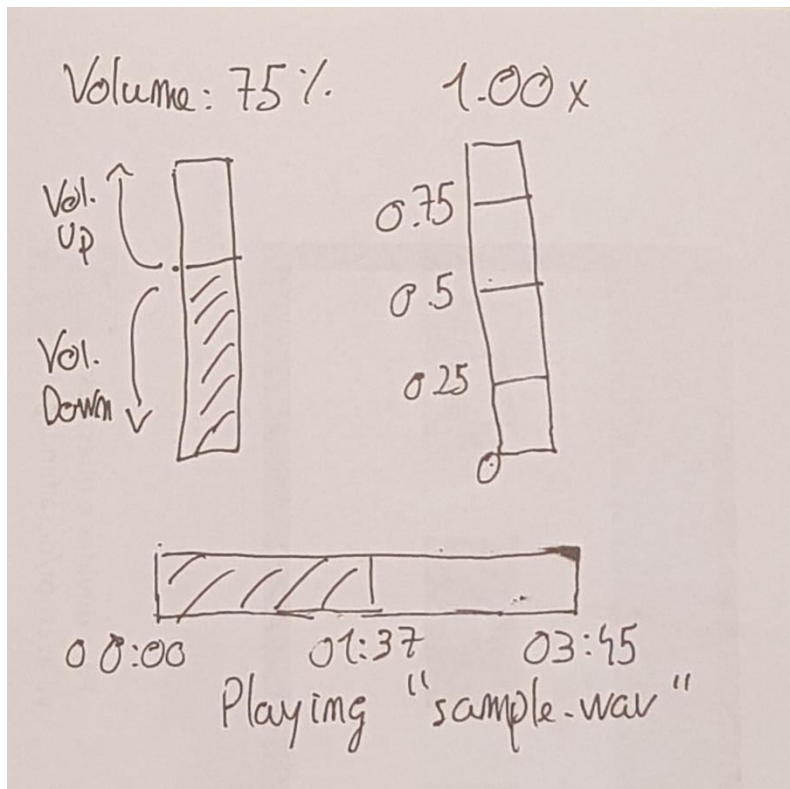


Figure 16 - Rough Sketch of *L-Music*'s Audio Manipulation Sliders

In Figure 16, the reader can visualize the initial design for *L-Music*'s audio manipulation sliders. These act on the generated musical fragments so that the user may hear them in their preferred way. The factor of component consistency here relies on the fact that only a slider is used, albeit in different ways as a way to convey their differences to the user:

- The volume slider is oriented vertically, and the user may slide the slider's track up and down in a continuous fashion from 0% (no volume) to 100% (maximum volume). The slider will fill up with a different color to visually inform volume changes to the user, besides having a text label on top of the slider.
- The playback speed slider is oriented vertically, and the user may slide the slider's track up and down in a discrete fashion in steps of 0.25, ranging from 0 (extremely slowed down audio) to 1.00 (regular speed). The slider is accompanied by text labels conveying each of the discrete values.
- The audio duration slider is oriented horizontally, and the user may slide the slider's track left and right in a continuous fashion to go to the beginning of the audio file or to the end of the audio file. The change in orientation informs the user that this slider differs from the other two, the change being that this slider does not directly affect how the audio will be perceived by them. Furthermore, the audio duration slider is accompanied by a starting label "00:00", indicating the beginning of the audio file; a label that accompanies the slider's track as the audio is played, in this case "01:37"; a label with the total duration of the audio file, in this case "03:45" and a label on the bottom of the slider informing the user what audio file is being played, in this case

“sample.wav”. Following suit to the volume slider, the audio duration slider will also fill up its track with a different color to provide a visual update to the user that an audio file is being played.

With only slight changes in the way each slider operates and is presented to the user, they provide different functionalities and convey different meanings¹⁶. However, their underlying aspects are all the same, that is to say, they are all either continuous or discrete sliders that manipulate audio in some way. The user will then memorize that slider components in *L-Music* are always related to audio manipulation features.

6.2.3.3 Creating a Custom L-System

The most complex feature in terms of user interactions is creating custom L-Systems. This is due to: (i) the user’s lack of knowledge on L-Systems. This is expected to happen given that the application is targeted at musicians and composers, who will rarely be in contact with generative formal grammars in their quotidian work; (ii) it requires a lot of user input that needs system validation as it is being given. The simplest L-System, which is deterministic and context-free, requires an alphabet of symbols; an axiom that is only made up of alphabet symbols and a rule set, where a predecessor and a successor with both also only being made up of alphabet symbols and (iii) depending on the type of L-System that is desired by the user, the inputs may differ (e.g., a Stochastic L-System requires a probability distribution function per production rule set with the same predecessor). This means that the system will need to adapt itself to request different types of inputs. Therefore, it is necessary to assume that the system will:

1. Facilitate the user into being familiar with L-Systems when creating them.
2. Validate every user input to ensure a low error rate.

¹⁶ The reader should note that using the appropriate copy for each slider also influences them having different meaning. If no text labels were associated with each slider, it would be much harder for a user to discern what each slider’s functionality was. Furthermore, having a common color scheme for sliders that represent audio manipulation functionalities is another factor that accentuates component consistency.

Figure 17 shows the initial design for the custom L-System form.

L-System Generator

1. Type

☒ Context-free ☐ Context-sensitive

☐ Stochastic ☐ Table

2. Alphabet

[A, B, C, D] → show error if not formatted properly

3. Axiom

[ABA] → show error if contains symbols & alphabet

4. Rule Set

show error if not formatted properly ← [B: AAB]

Only enable if form is valid

[CREATE]

+ Rule List

A: AB X
C: CC X
D: DA X

Figure 17 - Rough Sketch of the design for creating a custom L-System

A breakdown of Figure 17 follows. The form establishes a top-bottom visual hierarchy and flow, indicated by the “L-Systems Generator” title at the top as well as the numbered form fields with an explanatory label: “1. Type”; “2. Alphabet”; “3. Axiom”; “4. Rule Set”. On that account, an explanation of each form field is presented:

1. Form Field 1 requires the user to select the L-System type. It is expected that the system will adapt the remaining form fields as necessary given the chosen L-System type.

2. Form Field 2 requires the user to define the L-System's alphabet. The system also imposes on the user that they format the input correctly (in this case, separate each symbol with a comma). Otherwise, an error message will be shown. This increases the learning curve of the system, as a user with no previous L-System experience could expect just having to insert any symbols they would like.
3. Form Field 3 requires the user to define the L-System's axiom. The system here imposes a logical condition, that is, the axiom can only contain alphabet symbols. Otherwise, the user will see an appropriate error message.
4. Form Field 4 requires the user to define the L-System's production rule set. The system here imposes on the user that they format the input correctly (in this case, the input must follow the form "predecessor: successor". The system ignores any trailing or inner whitespaces). Otherwise, an error message will be shown. Furthermore, once the user is done defining a production rule, they need to add it to a list (pictured to the right of the input field) by tapping on the "+" button. The system allows the user to erase an added production rule in case they do not wish to have it anymore.

The final step for the user is to tap the "CREATE" button to add their L-System to the application's L-System list (pictured in Figure 14). This button is only enabled if the system confirms that the L-System form is in a valid state. Otherwise, it is disabled, giving visual information to the user that more action is required in order to create a custom L-System.

The contemplated design does not tackle some issues as well as it could. For example, to compensate a user's lack of knowledge in L-Systems, the application could offer the user a chance to view a brief explanation and demonstration of what L-Systems are and how they work. However, this can also be considered worse if we consider that we want L-Systems to remain a black box¹⁷ to the user and not impose any learning on the user's side¹⁸. Furthermore, a user might be more retentive in not using the system if they know a learning hurdle needs to be overcome.

6.3 Chapter Summary

This chapter served to document *L-Music's* intended design, both from a software architecture perspective and a Human-Computer Interaction (HCI) perspective. Section 6.1 catalogued the system and application's design via UML notation and diagrams. Section 6.2 reported the user interface and user experience design for *L-Music* as well as some concerns that were taken into consideration for the former.

¹⁷ A black box is typically considered something that receives input and transforms it into output without knowing anything about how the input is processed and transformed.

¹⁸ This can be overcome by making the L-System demonstration optional and not very intrusive for the user.

7 Implementation

This chapter documents the implementation of the *L-Music* prototype, an assistive tool for music composition based on L-Systems. Its outline follows. Section 7.1 concerns the implementation of the UI/UX proposed in section 6.2.3. Section 7.2 comprises of the implementation of the selected L-System types for the L-System module. Section 7.3 documents the implementation of the musical interpretations of L-Systems using the *sf2_loader* and *musicpy* libraries.

7.1 User Interface Module

The UI Module is implemented using the Flutter framework. This section aims to detail the implementations made regarding the proposed design in 6.2, as well as what diverged from it. Section 7.1.1 showcases how visual hierarchy and flow within the app were achieved. Section 7.1.2 goes over each core Widget¹⁹ of *L-Music*'s user interface. Section 7.1.3. documents how the UI conditions itself in face of user input (e.g., when selecting a different musical parser). The last section, 7.1.4, encompasses the implementation changes of creating a custom L-System versus the initial proposed design.

7.1.1 Visual Hierarchy and Flow

To establish a visual hierarchy between the various UI elements of *L-Music* as well as a flow of interactions for the user, the landing screen of the application is an empty instruments list, informing the user that they need to load a SoundFont file. This is illustrated by Figure 18 - Empty Instrument List in *L-Music*.

¹⁹ A Widget in Flutter is an abstraction of a User Interface element [151].

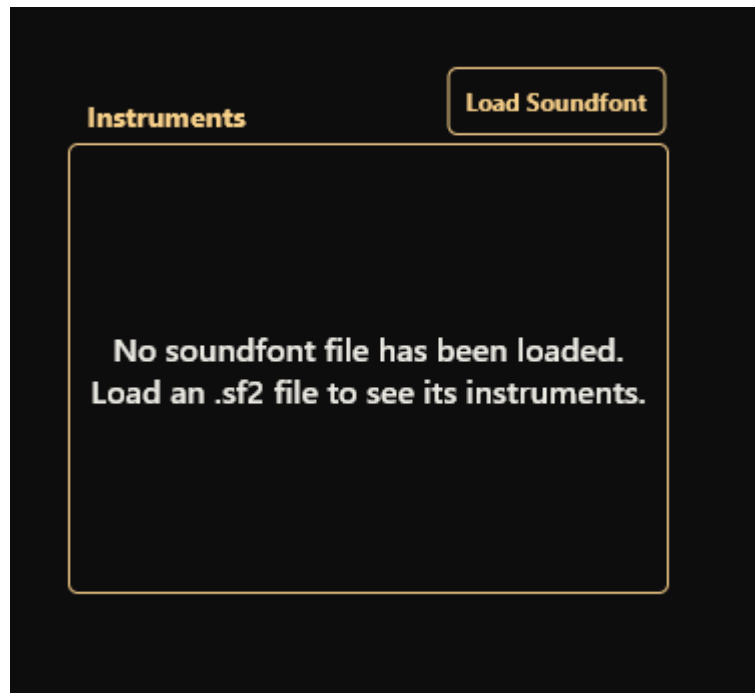


Figure 18 - Empty Instrument List in *L-Music*

By analyzing Figure 18, the reader is also reminded of the proposed color scheme for the application, present in section 6.2.1, which was implemented too. Furthermore, by presenting this UI element first, the user establishes the connection that it is (a) the starting point of their interaction with the system and (b) a core element for generating musical fragments. When the user taps the “Load Soundfont” button, they are prompted with a native dialogue to load a file onto *L-Music*. This is visible in Figure 19 .

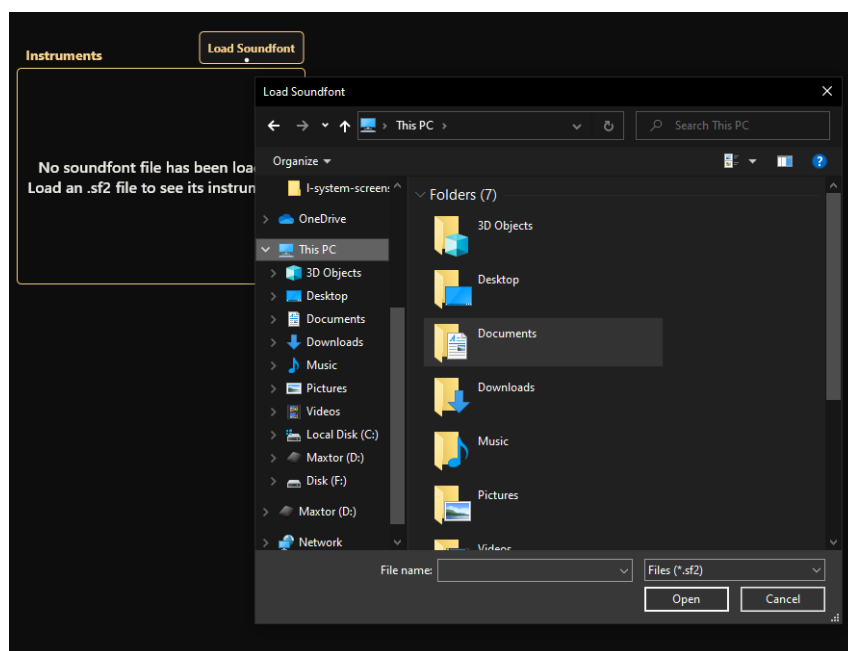


Figure 19 - Native Dialog to request a SoundFont File

Upon the user selecting the file and the file being loaded, the rest of the user interface appears through an animated fade in effect. Due to the interface size being large, it is broken up into sections in 7.1.2.

7.1.2 *L-Music* UI Breakdown

This section segments *L-Music*'s UI into its most important areas of interaction. These are as follows: the SoundFont Instrument List Widget; the L-System List Widget; the Musical Parser List Widget; the Widgets which make up the Musical Parser Parameters Section and the Audio Manipulation Section Widgets. The L-System generator section, for creating custom L-Systems, is tackled in section 7.1.4.

7.1.2.1 SoundFont Instrument List Widget

The SoundFont Instrument List Widget has two states: an empty state (which is in Figure 18) and a loaded state, visible in Figure 20.

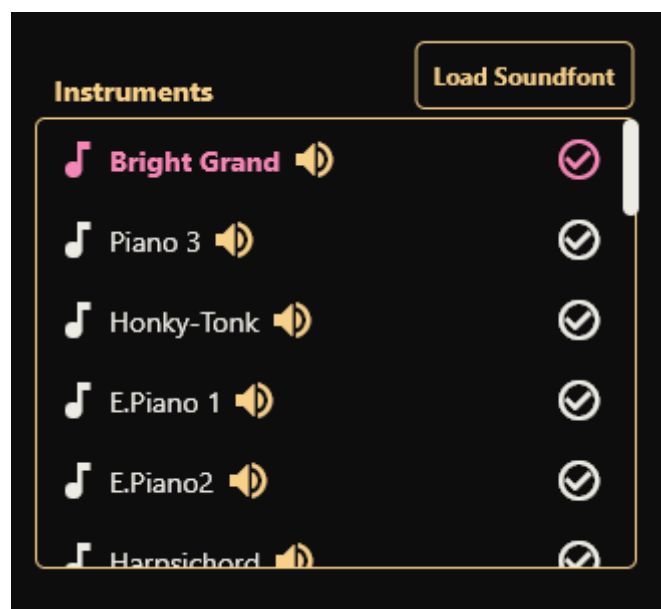


Figure 20 - SoundFont Instrument List - Loaded State

The list displays all instruments of the loaded file in the form of list items. Each instrument is therefore represented by:

- A note icon followed by the instrument's name. This further conveys to the user that the list item represents a musical instrument.
- A speaker Icon Button. By tapping this, the user can hear a predefined sample of the instrument to get acquainted with its texture quality²⁰.

²⁰ The reader can listen to this sample at <https://soundcloud.com/user-62526899/l-music-app-instrument-sample>.

- A circled checkmark icon, to serve as a visual aid that the items of this list are selectable. In this case, tapping the whole item, with the exception of the speaker icon button, will select the instrument. In the case of Figure 20, the instrument *Bright Grand* is selected.

7.1.2.2 L-System List Widget

The L-System List Widget was implemented in a similar fashion to the instrument list in order to achieve component consistency. It is displayed in Figure 21.

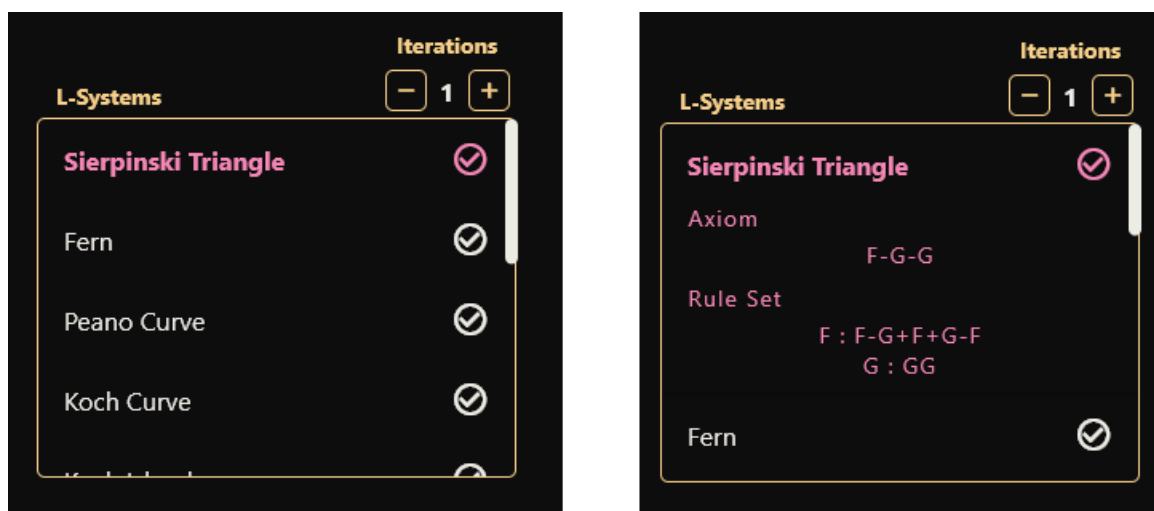


Figure 21 - The L-System List. On the left, the selected item is in a collapsed state. On the right, it is in an expanded state, showing details about the "Sierpinski Triangle" L-System, namely its axiom and production rule set. On the top is an iteration counter for the user to indicate how many generations of the selected L-System are desired.

The L-System list differs slightly from the instrument list. Since an L-System has additional information (not just a name like a musical instrument), it was necessary to utilize ExpansionTile Widgets, in order to hide and show that information appropriately. The user can also encounter an iterations counter at the top of the list. This counter, as its label suggests, represents the number of iterations for the selected L-System. Due to the necessity of having a touch area for the whole list item in order to expand it and collapse it, the touch area for selecting an L-System is different than for selecting an Instrument. Figure 22 showcases this.

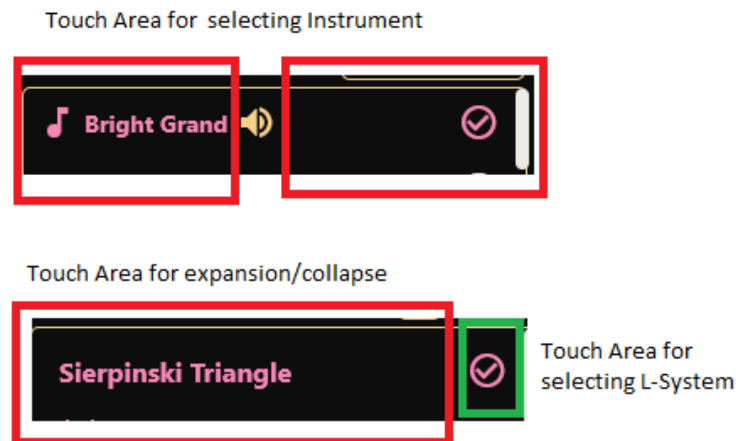


Figure 22 - Touch Area differences between an Instrument List Item and an L-System List Item

7.1.2.3 Musical Parser List Widget

The Musical Parser List Widget was implemented in a similar way to the L-System List Widget. Again, this allows to achieve a consistent component usage across *L-Music*. It is shown in Figure 23.

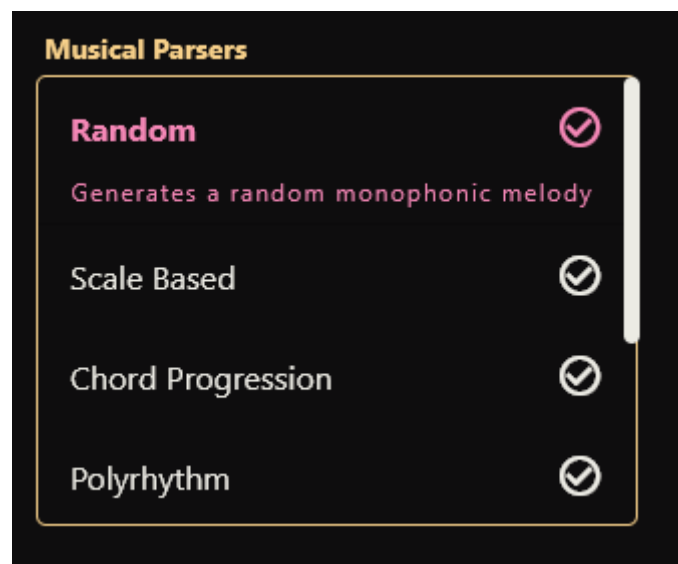


Figure 23 - Musical Parsers List

The list items of the Musical Parsers list follow the same behavior as the ones of the L-System List. They have an expanded and collapsed state. When expanded, the user can read a brief description of the type of excerpt the musical parser will generate.

These three lists are followed by each other, starting with the Instrument list, the L-System list, and the Musical Parser list (see X). They stand atop the remaining components of the UI, with the exception of the audio manipulation components. This reinforces the notion of establishing a visual hierarchy and flow for the user in order to simplify their experience.

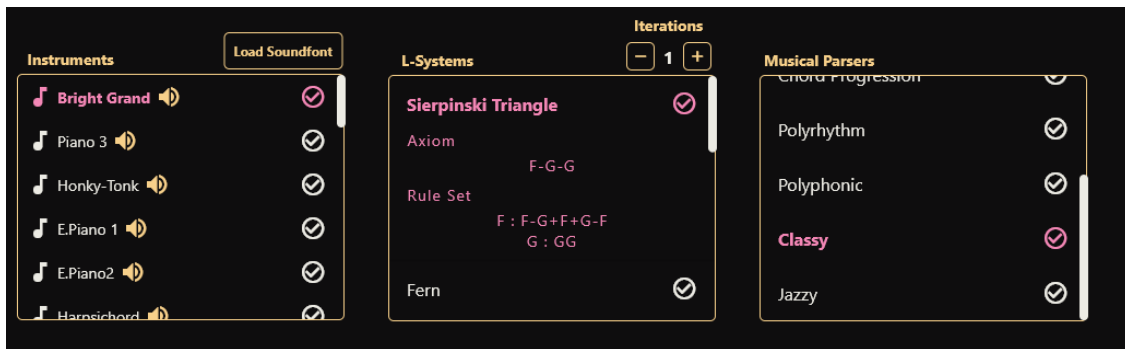


Figure 24 - The top section of *L-Music*, consisting of the *Instruments*, *L-Systems* and *Musical Parsers* Lists.

7.1.2.4 Musical Parser Parameters Section Widgets

The Musical Parser Parameters Section Widgets are a set of components which allow the user to input certain configurations for their desired musical fragment. The configurable parameters are: (a) scale; (b) rhythm; (c) key; (d) beats per minute (BPM); (e) root pitch and (f) number of octaves.

The scale parameter is represented by a list, comprised of various musical scales (see Figure 25.).

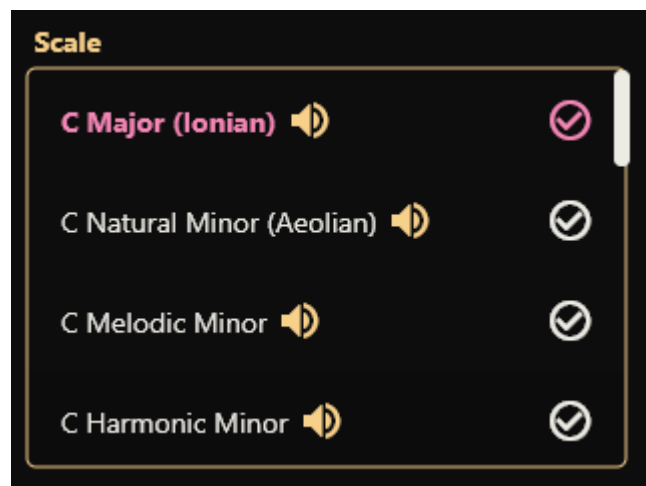


Figure 25 – Scale List

The list items of the Scale list follow the same implementation of the instruments list item, with the exception that the Scale list items do not have an icon on the left. This is due to the Scale name being the most important aspect of the list item, so there is no need for extra visual aid. The user can also click the speaker Icon Button to hear a scale being played with their selected instrument²¹.

²¹ The reader can listen to a scale sample at <https://soundcloud.com/user-62526899/l-music-app-scale-sample-c-harmonic-minor?si=b555445e006143e6809222b8541569c9>

The rhythm parameter is also represented by a list. It holds values related to rhythmic figures in music, such as the quarte note or the eighth note (see Figure 26).

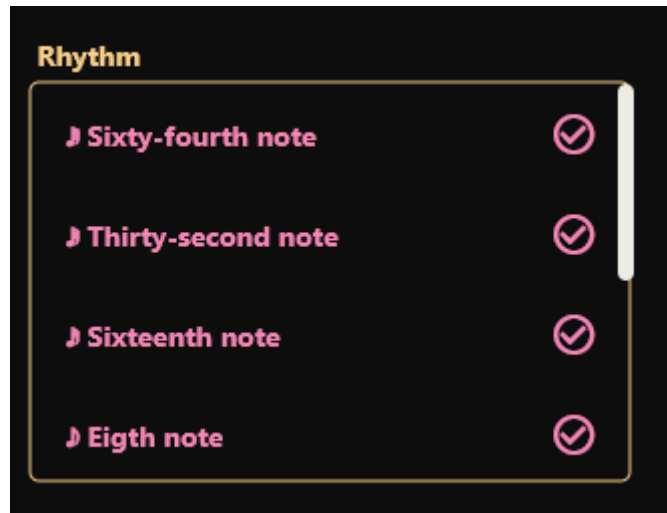


Figure 26 - Rhythm List

A rhythm list item is the simplest of all the list items. It contains the name of the rhythm figure, associated to its symbol, and a circled checkmark icon to indicate whether the item is selected or unselected.

The key parameter is represented by a dropdown menu. Its values consist of musical keys, like E Major (see Figure 27).

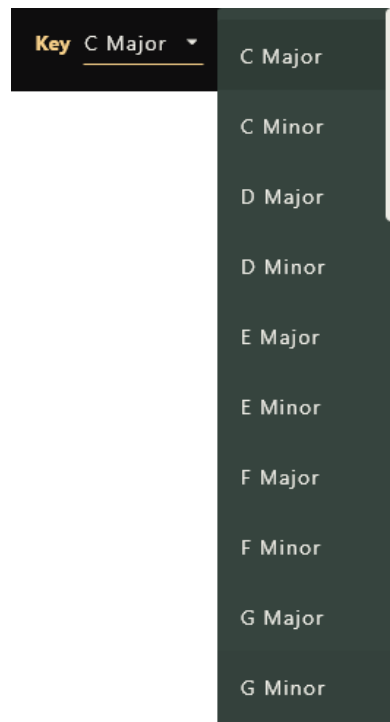


Figure 27 – Key Dropdown and Dropdown Menu

The BPM parameter is represented by a counter (see Figure 28). It has two buttons, for decreasing and increase the number of BPM, which allow a long hold action and a tap action. The long hold action will increase the number of BPM at a faster rate.

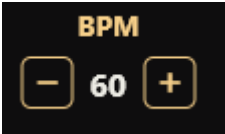


Figure 28 - Beats Per Minute Counter

The root pitch parameter is represented by a dropdown and dropdown menu (see Figure 29). Its values are representations of pitch across 11 octaves (e.g., C5 or D4).

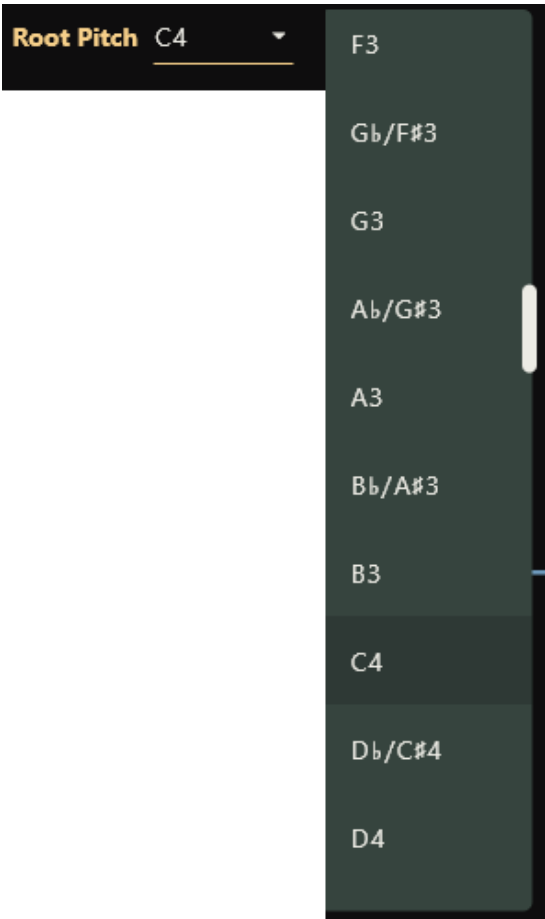


Figure 29 - Root Pitch Dropdown and Dropdown Menu

The number of octaves parameter is represented by a counter (see Figure 30), following the same design as the BPM counter.

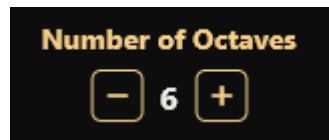


Figure 30 - Number of Octaves Counter

Besides containing all of the mentioned parameters, the Parser Parameter Section also contains the “Generate Music” Button (see Figure 31). It stems that if the user has finished configuring all possible parameters for a given Musical Parser, they will want to generate a new musical fragment. Therefore, the generate music button should be visually close to these parameters.



Figure 31 - The Generate Music Button. On the left, it is in its active state. On the right it is in a disabled state, showing a progress indicator on the right to inform the user that the operation is still being performed.

Figure 32 showcases the closeness principles between the parser parameters and the Generate Music Button.

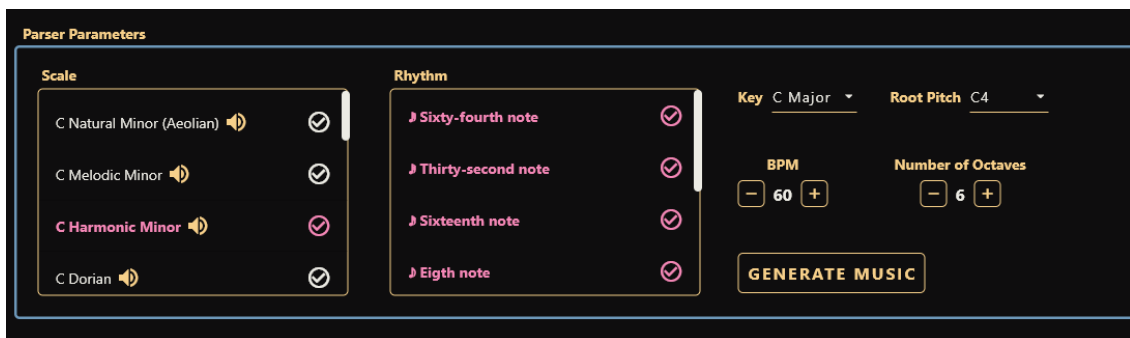


Figure 32 - Parser Parameters Section.

7.1.2.5 Audio Manipulation Section Widgets

The Audio Manipulation Section Widgets are the group of components that relay any audio related functionality to the user (e.g., playback speed or saving audio to a MIDI file). These components are: (a) the previous, play, next and pause buttons; (b) the save audio as WAV and MIDI buttons; (c) the volume, playback, and audio duration sliders.

The set of buttons to control audio files are presented in Figure 33. In order, from left to right, they are: the previous button, which is only enabled if the user has a playlist of previously listened to audio fragments during their session; the play button, which will begin playing the queue’s current audio file; the next button, which is only enabled if the user has a playlist of

remaining audio fragments to listen to during their session and the pause button, which will pause the currently played audio file.



Figure 33 - Audio Playback Buttons

Next, the save audio as WAV and MIDI file buttons, in Figure 34.



Figure 34 - Save audio as WAV or MIDI File buttons

Upon tapping either, the user is prompted with a native dialog to select a path and file name for their WAV or MIDI files. Upon a successful save, *L-Music* will relay that information by means of a Snackbar [152], visible in Figure 35.

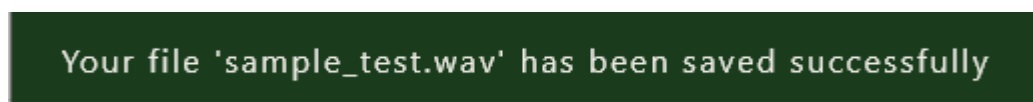


Figure 35 - Example of a Snackbar in *L-Music*. In this case, the system is informing the user that a file named "sample_test.wav" was saved successfully in the user's device.

Figure 36 displays the sliders that allow the user to control some properties of an audio that is currently being played.

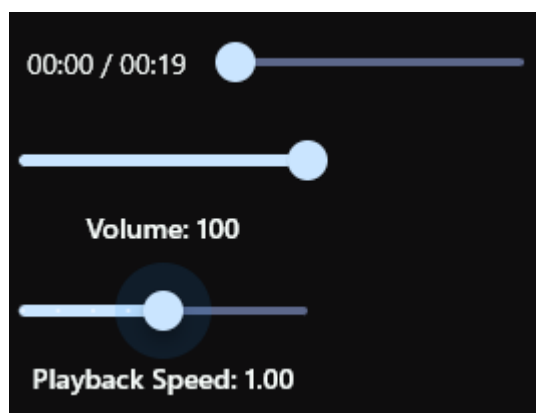


Figure 36 - *L-Music* Audio Sliders. In order, from top to bottom: audio position slider, audio volume slider and audio playback speed slider.

While the audio position slider merely provides the user a way to jump between parts of an excerpt, the volume and playback speed sliders actually affect the user's listening experience (e.g., an excerpt might not sound interesting at its normal speed but may offer a different

perspective once slowed down or sped up, or if it's quieter or louder). For this reason, both sliders are kept together while the audio position slider is closer to the audio buttons in *L-Music's* interface (see Figure 37).

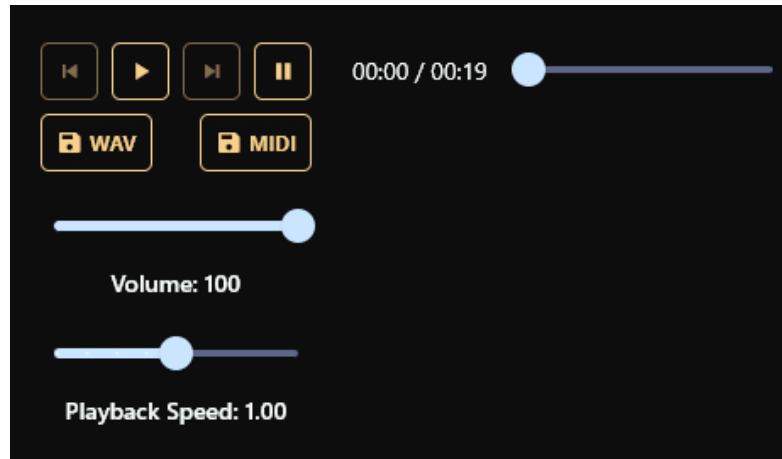


Figure 37 - Component Layout of the Audio Widgets Set

7.1.3 Conditioning User Input

Conditioning user input, meaning that restricting what the user may or may not edit under certain conditions is important given that the system should avoid asking for information that is not required for a requested interaction. This is exemplified by limiting the parser parameters the user may configure based on the selected Musical Parser. Table 3 maps out what parser parameters are configurable per Musical Parser type.

Table 3 - Musical Parser Configurability. "NC" stands for Non-Configurable. "C" stands for Configurable. Row names represent Musical Parser Types. Column names represent parser parameters.

| | Scale | Rhythm | Key | BPM | Number of Octaves | Root Pitch |
|-------------------|-------|--------|-----|-----|-------------------|------------|
| Random | NC | NC | NC | NC | NC | NC |
| Scale Based | C | NC | NC | C | C | C |
| Chord Progression | C | C | NC | C | NC | NC |
| Polyrhythm | NC | C | NC | C | C | C |
| Polyphonic | C | NC | NC | C | C | C |
| Classy | C | C | C | C | C | C |
| Jazzy | C | C | C | C | C | C |

To convey this information visually, the parser parameters turn opaque to indicate that they are non-actionable given the selected Musical Parser. An example of the former is in Figure 38.



Figure 38 - Example of conditioning user input. All parser parameters are opaque given that the selected parser is "Random"

7.1.4 Creating a Custom L-System

One of the most interesting use cases of *L-Music* is creating custom Lindenmayer Systems. This gives full extensibility and control of *L-Music*'s capabilities to the user since the musical fragments are generated from an L-System's output. Creating an L-System is very much like filling out a form. In the initial design of this use case (see 6.2.3.3), a vertical layout was proposed, to ensure a simple read through of the form. However, due to the horizontal stacked layout that the other sections of *L-Music* have (see Figure 38 for example), this design was changed in the development of the prototype to follow a horizontal layout and was inserted as the "third" layer of *L-Music*'s interface. Figure 39 showcases this.



Figure 39 - The stacked layout of *L-Music's* interface

The interface allows to easily identify three distinct sections of the application. Furthermore, it also allows the definition of their importance, being that the first section (the one comprising of the instruments, L-Systems, and musical parsers list) is the most important and the last section (the one comprising of the L-System generator) is the least important. This order stems from the fact that the user can do the majority of their work without ever interacting with the Parser Parameters section (excluding the generate music button) and the L-System Generator section. Since they are more advanced features, the user may not be interested in interacting with them at all. A detailed explanation of the L-System generator implementation follows.

The L-System generator subsection comprises of: (i) a dropdown where the user can select the L-System type; (ii) four text fields, for adding info about the L-System itself; (iii) a list view, where the user can see and remove rules from the L-System under creation and (iv) a button to add the L-System to the L-Systems list.

The dropdown for the L-System type follows the same behavior as previously mentioned ones (see, for example, Figure 29). The user is allowed to select from four types of L-Systems, as seen in Figure 40.

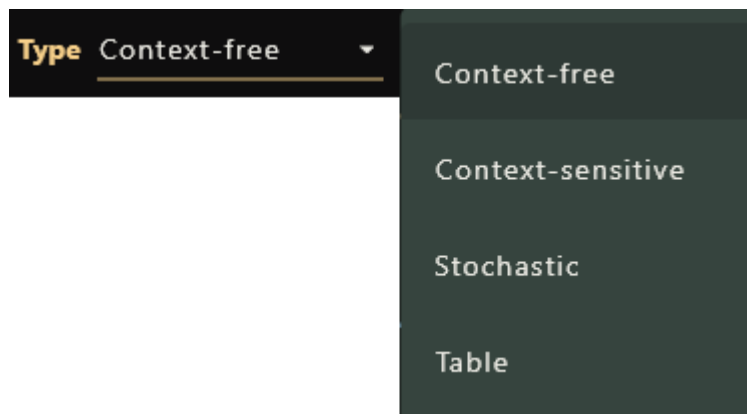


Figure 40 - L-System type Dropdown and Dropdown Menu

The focus is now on the text fields of the form. The user needs to input the following via the four text-fields:

- L-System name. This is so that the user can later identify them in the L-Systems list.
- The L-System's alphabet. This is a list of symbols that need to be separated by commas (e.g., "A,B").
- The L-System's axiom. This needs to be a sequence of symbols from the alphabet.
- The L-System's production rule set. The rules need to follow the format "predecessor: successor", where predecessor is a symbol from the alphabet and predecessor is a sequence of symbols from the alphabet.

Figure 41 displays all four text fields.

Figure 41 - Text Fields for creating an L-System in *L-Music*

The reader should note in Figure 41 that each text field is supplied with a textual hint. This is to provide some sort of guidance and help to users who may not be familiar with L-Systems. Furthermore, given that the alphabet, axiom, and rule set fields require validation, the system needs to promptly inform the user of any errors. Figure 42 illustrates how the system reacts if the user inputs a mal-formatted alphabet.

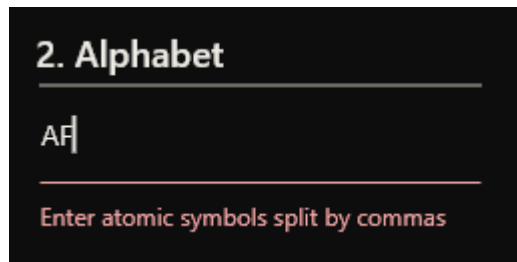


Figure 42 - Error Message for a mal-formatted L-System alphabet when creating an L-System

By using an appropriate copy for the error message in Figure 42, the system can transmit to the user what actions need to be taken to input an L-System's alphabet in the correct format. The same happens for the axiom's validation, visible in Figure 43.

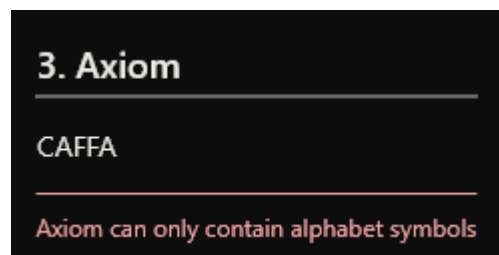


Figure 43 - Error Message for an invalid axiom when creating an L-System. In this case the defined alphabet only contains the symbols "A" and "F"

In Figure 43, the user defined an alphabet consisting of symbols "A" and "F". As such, the system identifies a symbol that is not part of the alphabet in the axiom field and warns the user of the issue appropriately. The same behavior happens for production rule validation, visible in Figure 44.

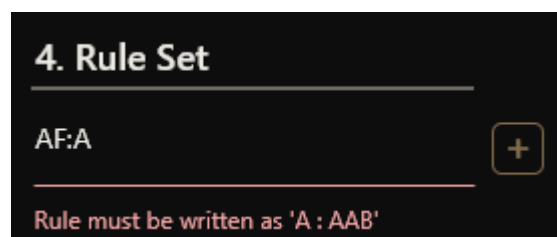


Figure 44 - Error Message for an invalid production rule when creating an L-System

In Figure 44, the system identified that the predecessor of the rule is not of length 1, and as such warns the user promptly that the rule needs to have a length 1 predecessor. The reader should see, in the same Figure, that there is a "+" Icon Button to the right of the text field. Because an L-System can have multiple rules, the system needs to manage the latter. This is achieved by using a list view for the rule set, illustrated by Figure 45.

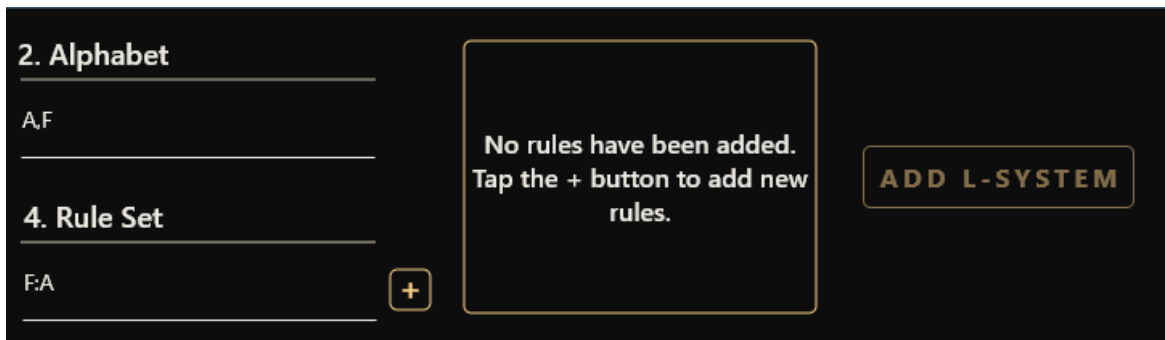


Figure 45 - Empty rule set list view when creating an L-System

In Figure 45, no rules were added to the L-System's set yet. As such, the system informs the user of the fact and communicates via text that they should press the "+" button to add new rules. When the user does so, the list will look like the one in Figure 46. Additionally, the "Add L-System" button is disabled, as the L-System is not in a valid state yet (i.e., no production rule set has been defined).

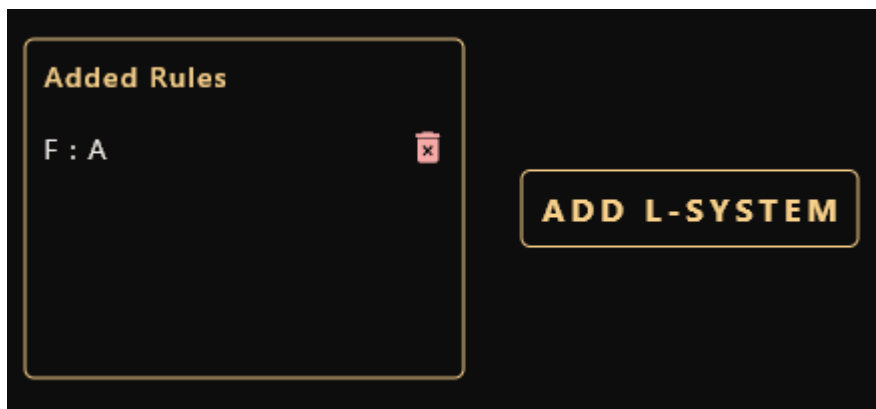


Figure 46 - Rule set list view after a rule is added

As seen in Figure 46, the rule set list view contains all rules added by the user when creating an L-System. Its list items contain two elements: the rule itself and a trash icon. The latter serves as a visual call to a delete action in case the user wants to remove a production rule. As long as the L-System is validated, the "Add L-System" button will be enabled, and the user will be able to add their custom L-System. It will appear in the L-Systems list, as exemplified by Figure 47.



Figure 47 - Example of a custom L-System list item

A custom L-System's list item has added functionality when compared to the default L-Systems. In Figure 47, the reader will be able to see a "Delete L-System" text below the L-System's description. This allows the user to remove any unwanted L-Systems (e.g., the created L-System does not provide the desired musical results).

7.2 L-System Module

This section documents the implementations of the selected L-System types: (a) deterministic and context-free; (b) deterministic and context-sensitive; (c) stochastic, or non-deterministic and (d) table L-Systems. The implementations were done using the Python programming language. For formal definitions of the mentioned L-System types, the reader should refer to sections 5.2.1, 5.2.2, 5.2.3 and 5.2.7 respectively.

The reader should consider the classes present in Code Snippet 1 to serve as the basis of all implementations, considering the documented design in 6.1.2.2.

Code Snippet 1- Base classes for the implementation of the L-System module

```
class LSystem(ABC):
    def __init__(self, axiom, rules):
        self.axiom = axiom
        self.rules = rules

    @abstractmethod
    def generate(self, iterations=1):
        pass

class Rule:
    def __init__(self, predecessor, successor):
        self.a = predecessor
        self.b = successor
```

An explanation of Code Snippet 1 follows. The *LSystem* class is abstract (i.e., the reader should notice the “ABC” tag), and has two attributes, *axiom*, and *rules*, which, as the names suggests, represents the needed properties of an L-System in order to implement the method *generate*. The latter is also abstract, since the implementation logic will be dependent on the type of L-System, and it receives a number of iterations to know how many times the L-System’s parallel string rewriting mechanism should be applied. Furthermore, the reader should also assume that the majority of operations are performed on string objects, given that L-Systems rely on parallel string rewriting.

7.2.1 DOL-Systems (Deterministic, Context-Free)

The implementation of DOL-Systems serves as the basis for the other types due to the fact that the parallel string rewriting mechanism varies only in certain aspects (e.g., in context-sensitive systems, one must have the concern of validating the contexts of a predecessor before rewriting its successor onto the final result). Code Snippet 2 showcases the implementation of the *generate* method for a DOL-System.

Code Snippet 2 - Implementation of a DOL-System

```
class DolSystem(LSystem):
    def generate(self, iterations=1):
        sentence = self.axiom
        if(iterations == 0):
            return self.axiom
        while(iterations != 0):
            nextGeneration = ""
            for i in range(len(sentence)):
                currentSymbol = sentence[i]
                replace = "" + currentSymbol
                for j in range(len(self.rules)):
                    predecessor = self.rules[j].predecessor
                    if(predecessor == currentSymbol):
                        replace = self.rules[j].successor
                        break
                nextGeneration += replace
            sentence = nextGeneration
            iterations -= 1
        return nextGeneration
```

An explanation of Code Snippet 2 follows. A *DolSystem* class is defined, which extends the *LSystem* class, and as such must override the *generate* method. The latter is as follows. If the requested number of iterations is zero, the method may simply return the L-System's axiom. Otherwise, it will iterate over the desired number of iterations. The reader is also pointed out to the declaration of the variable *sentence*, which starts by containing the L-System's axiom. *nextGeneration* is responsible for holding the result of the appliance of the parallel string rewriting mechanism. For each symbol of the axiom, the method will iterate the L-System's production rule set. It will also declare a variable named *replace*, which represents the symbol that might be rewritten or not. Due to the latter, its default value is *currentSymbol*. Per rule, the following check is performed: if the rule's predecessor matches the current symbol of the axiom that is being visited (represented by the variable *currentSymbol*), *replace* will be assigned the value of that rule's successor. *sentence* is assigned the value of *nextGeneration*, as a means of ensuring that the string rewriting happens on the most recent output. *nextGeneration* is concatenated with the value of *replace* once all rules have been iterated over for a given symbol of *sentence* and is therefore the desired return value.

7.2.2 DIL-Systems (Deterministic, Context-Sensitive)

DIL-Systems are context-sensitive, and as such have a different mechanism for validating whether production rules are applied as well as having different rules. As such, an implementation of a *DilRule* is presented in Code Snippet 3.

Code Snippet 3 - Implementation of a DIL production rule

```
class DilRule(Rule):
    def __init__(self, leftContext, predecessor, rightContext, successor):
        self.leftContext = leftContext
        self.predecessor = predecessor
        self.rightContext = rightContext
        self.successor = successor
```

The *DilSystem* class, as well as its *generate* override follows, in Code Snippet 4.

Code Snippet 4 – Implementation of a DIL System

```
class DilSystem(LSystem):
    def generate(self, iterations=1):
        sentence = self.axiom
        if(iterations == 0):
            return self.axiom
        while(iterations != 0):
            nextGeneration = ""
            for i in range(len(sentence)):
                currentSymbol = sentence[i]
                replace = "" + current
                for j in range(len(self.rules)):
                    rule = self.rules[j]
                    leftContext = rule.leftContext
                    predecessor = rule.predecessor
                    rightContext = rule.rightContext
                    if(currentSymbol == predecessor):
                        sentenceLeftContextAux = sentence[0: i]
                        sentenceLeftContext = sentenceLeftContextAux[-len(leftContext):]
                        sentenceRightContext = sentence[i+1: i + 1 + len(rightContext)]
                        isLeftContextEmpty = leftContext == '0'
                        isRightContextEmpty = rightContext == '0'
                        isLeftContextValid = sentenceLeftContext == leftContext or isLeftContextEmpty
                        isRightContextValid = sentenceRightContext == rightContext or isRightContextEmpty
                        if(isLeftContextValid and isRightContextValid):
                            replace = rule.successor
                nextGeneration += replace
            sentence = nextGeneration
            iterations -= 1
        return nextGeneration
```

An explanation of the *generate* override follows. We will only concern with detailing the mechanism for validating if the string rewriting should happen, given that the underlying implementation is the same as the one found in Code Snippet 1. Once *currentSymbol* matches a production rule's predecessor, the following logic is applied:

- First, the right and left context of *sentence* relative to the current rule's *leftContext* and *rightContext* and *currentSymbol* are extracted (i.e., *sentenceLeftContext* and *sentenceRightContext* respectfully). A simple example for clarification is given: assuming that *sentence* is "AABFBCC", that *currentSymbol* is "F" and that *leftContext* is "A" and *rightContext* is "C", then *sentenceLeftContextAux* is "AA", *sentenceLeftContext* is "A" and *sentenceRightContext* is "B".
- Second, a check for empty contexts is made (i.e., *isLeftContextEmpty* and *isRightContextEmpty*). This required the reservation of the symbol "0" (i.e., zero or nil) to make the comparison feasible²².
- Third, a check is performed to validate whether the *leftContext* and *rightContext* of the current production rule are being met.
- Fourth, if both contexts are valid (i.e., both *leftContext* and *rightContext* were found relative to *currentSymbol*'s position in *sentence*), the successor of the current production rule is assigned to *replace*, and the string rewriting mechanism is applied.

7.2.3 Stochastic L-Systems (Non-Deterministic)

Much like DIL-Systems, Stochastic L-Systems also require their own type of production rule. Code Snippet 5 displays the implementation for a Stochastic production rule:

Code Snippet 5 - Implementation of a Stochastic production rule

```
class StochasticRule(Rule):
    def __init__(self, predecessor, successorProbabilityDistribution):
        self.predecessor = predecessor
        self.successor = successorProbabilityDistribution
```

For simplicity purposes, we assume that *successorProbabilityDistribution* is a dictionary, or map, object which associates a successor to a probability. Code Snippet 6 exemplifies a *StochasticRule* object.

Code Snippet 6 - Example of a *StochasticRule* object

```
StochasticRule('a', {'ab': 0.7, 'ba': 0.3})
```

The implementation for the *StochasticSystem* class is presented in Code Snippet 7. Much like for DIL-Systems, we will only concern with detailing the validation for applying string rewriting.

²² It is also possible to argue that the symbol which defines an empty context could be defined by the user. This implies that besides creating L-Systems, the user is also able to dictate what meaning each symbol has.

Code Snippet 7 - Implementation of a Stochastic L-System

```
import random
import sys

class StochasticLSystem(LSystem):
    def generate(self, iterations=1):
        sentence = self.axiom
        if(iterations == 0):
            return self.axiom
        while(iterations != 0):
            nextGeneration = ""
            for i in range(len(sentence)):
                currentSymbol = sentence[i]
                replace = "" + currentSymbol
                for j in range(len(self.rules)):
                    rule = self.rules[j]
                    predecessor = rule.predecessor
                    if(predecessor == currentSymbol):
                        seed = random.randrange(sys.maxsize)
                        rng = random.Random(seed)
                        number = rng.random()
                        lowerBoundary = 0
                        upperBoundary = 0
                        for successor, probability in rule.successor.items():
                            lowerBoundary += probability
                            if(number < lowerBoundary and number >= upperBoundary):
                                replace = successor
                                upperBoundary += lowerBoundary
                        nextGeneration += replace
                sentence = nextGeneration
                iterations -= 1
            return nextGeneration
```

Once *currentSymbol* matches a production rule's predecessor, the following logic follows:

- A random number is generated from a random seed²³.
- For each probability distribution mapping, a check is performed to validate if the probability of the production rule is lower and higher than given boundaries (i.e., *lowerBoundary* and *upperBoundary*). The operations perform on the latter are to ensure that the distributions of the production rule mapping are met (i.e., if the distribution between two rules "A" and "B" is 70% to 30% respectively, on average, "A" should be chosen 70% of the time while "B" should be chosen 30% of the time.²⁴

²³ A seed is any number used to initialize a random number generator (RNG).

²⁴ The reader should note that any implementation is valid as long as the probability distribution of the production rule is met. One may desire, for example, to use statistical functions or other methodologies.

- If the previous condition is met, the given probability's *successor* is chosen to be applied to the string rewriting mechanism.

7.2.4 Table L-Systems

Table L-Systems define what production rule tables to use based on a condition. For our implementation, we assumed that this condition would be:

$$currentIteration \leq originalIteration - 2$$

If the condition proves true, the table with name, for example, *table2* is used. Otherwise, a table with name, for example, *table1* is used.

For simplicity purposes, we assumed that a production rule table could be modelled after a production rule. However, the predecessor field is a dictionary, representing the set of production rules associated with the table's name, or identifier. An example is in Code Snippet 8.

Code Snippet 8 - Example of a Production Rule Table

```
{'table1': [Rule('a', 'ab'), Rule('b', 'a')], 'table2': [Rule('a', 'b'), Rule('b', 'ba')]}
```

The implementation for a Table L-System follows in Code Snippet 9. As was the case for Stochastic and DIL-Systems, we will only concern with detailing the context under which the string rewriting is in.

Code Snippet 9 - Implementation of a Table L-System

```
class TableLSystem(LSystem):
    def generate(self, iterations=1):
        sentence = self.axiom
        original = iterations
        if(iterations == 0):
            return self.axiom
        while(iterations != 0):
            nextGeneration = ""
            rulesToUse = self.rules['table2'] if iterations <= (
                original - 2) else self.rules['table1']
            for i in range(len(sentence)):
                currentSymbol = sentence[i]
                replace = "" + currentSymbol
                for j in range(len(rulesToUse)):
                    predecessor = rulesToUse[j].predecessor
                    if(predecessor == currentSymbol):
                        replace = rulesToUse[j].successor
                        break
                nextGeneration += replace
            sentence = nextGeneration
            iterations -= 1
        return nextGeneration
```

In this manner, an explanation of Code Snippet 9 is provided. Every iteration step, a check is made to see if the Table L-System's condition is verified or not. Depending on that, the production rule tables are switched or not. In our implementation, for 5 iterations, the tables would switch on iteration step 3.

7.2.5 Considerations regarding L-System Implementation

The reader should note that each L-System was implemented to only handle one type of rule. However, this can be easily extended to handle several types of rules by type checking the *Rule* objects. This way, it is possible to achieve Stochastic Context-Sensitive L-Systems or Stochastic Table L-Systems with relative ease.

7.3 Musical Parser Module

This section documents the implementation of the Musical Parser Module, responsible for manipulating SoundFont files as well as interpreting L-Systems musically. First, we acquaint the reader with the *sf2_loader* and *musicpy* libraries which greatly aided the implementation of this module. Then, we go over each of the implemented musical interpretations for L-

Systems: (i) random interpretation; (ii) scale-based interpretation; (iii) chord progression interpretation and (iv) polyphonic interpretation.

7.3.1 Showcasing the `sf2_loader` and `musicpy` libraries

The `sf2_loader` library exposes an API to handle SoundFont files. Code Snippet 10 showcases some of the operations that the library allows, such as loading a file and getting its list of instruments.

Code Snippet 10 - `sf2_loader` library usage example

```
import sf2_loader as sf

sf2 = sf.sf2_loader("sf2FilePath")
allInst = sf2.get_all_instrument_names(track=None,
                                       sfid=None,
                                       bank_num=None,
                                       num=0,
                                       get_ind=False,
                                       mode=0)
```

The `musicpy` library exposes an API to write music via code. It abstracts music theory concepts, such as scales, chords, keys, modulation, and others. It offers several layers of complexity to accommodate users who may not be very familiar with the latter. Code Snippet 11 showcases some of the operations that the library allows, such as creating a scale with a name and mode, creating a note based on its MIDI pitch, creating a chord, and creating a *piece*, which for the library means any collection of chords or notes.

Code Snippet 11 - `musicpy` library usage example

```
import musicpy as mp

scale = mp.scale('C', 'Locrian')

# 60 translates to the note C4
noteFromMidiPitch = mp.degree_to_note(60)

# This chord represents the C Major Triad chord
chord = mp.chord(['C', 'E', 'G'])

# This piece will comprise of the C Major Triad chord, played by
# the instrument at preset 1 (typically it is a Piano)
piece = mp.piece([chord], [1])
```

7.3.2 Interpreting L-Systems Musically

This section documents the implementation of the musical interpretations of L-Systems. At the beginning of implementation, there were eight interpretations in mind. These were:

- A random interpretation. As the name suggests, the interpretation is purposefully random. Every decision that this interpretation makes with regards to what notes to play, how loud to play them, how long each note last is random. Its implementation is laid out in section 7.3.2.1.
- A scale-based interpretation. As the name suggests, the interpretation is based on a musical scale. Its implementation is laid out in section 7.3.2.2.
- A chord progression interpretation. As the name suggests, the interpretation generates chord progressions. Its implementation was not possible due to time constraints.
- A polyrhythmic interpretation. As the name suggests, the interpretation generates excerpts that rely on using various rhythmic patterns and figures. Its implementation was not possible due to time constraints.
- A polyphonic interpretation. As the name suggests, the interpretation generates musical fragments that have two or more voices (i.e., two or more instruments). Its implementation, which was branched into two approaches, was done for two voiced fragments, and is laid out in section 7.3.2.3.
- A “classy” interpretation. This interpretation would rely on 18th Century Western Classical Music guidelines and concepts to generate its musical excerpts. Its implementation was not possible due to time constraints.
- A “jazzy” interpretation. This interpretation would rely on guidelines and concepts from jazz music to generate its musical excerpts. Its implementation was not possible due to time constraints.

Since the chord progression, polyrhythmic, classy, and jazzy interpretations were not implemented, they are considered future work of this project (see section 9.2).

Some symbols were assumed reserved for musically interpreting L-Systems. Namely:

- “F”, which means “play note”.
- “+” which means “increase note pitch by a given number of semitones”
- “-” which means “decrease note pitch by a given number of semitones”
- “G” which means “play note for second voice” in one of the implemented approaches for polyphonic interpretation.

Each implemented interpretation will also be accompanied by a musical point of view on why certain decisions were made. Furthermore, the reader will be invited to listen to multiple generated samples from each type of implemented interpretation. The reader is encouraged to listen to them as they read the remaining sections.

7.3.2.1 Random Interpretation

The implementation for the random interpretation is presented in Code Snippet 12.

Code Snippet 12 - Implementation of the random musical interpretation of an L-System

```
def random_note_pitch(lowestPitch=21, highestPitch=127):
    return random.randint(lowestPitch, highestPitch)

def random_parsing(output, noteDuration=1/8, lowestPitch=21, highestPitch=127,
    durations=[1/32, 1/16, 1/8, 1/4, 1/2, 1, 2, 4], lowestVolume=0,
    highestVolume=100):
    randomChord = []
    for i in range(len(output)):
        currentSymbol = output[i]
        duration = noteDuration
        pitch = random_note_pitch(
            lowestPitch=lowestPitch, highestPitch=highestPitch)
        newNote = mp.degree_to_note(pitch, duration=random.choice(durations))
        for j in range(1, len(output)):
            nextSymbol = output[j]
            if(currentSymbol == nextSymbol and currentSymbol == 'F'):
                duration += random.choice(durations)
            if(currentSymbol != nextSymbol and currentSymbol == 'F'):
                volume = random.randint(lowestVolume, highestVolume)
                newNote = mp.degree_to_note(
                    pitch, duration=duration, volume=volume)
                randomChord.append(newNote)
                duration = noteDuration
                pitch = random_note_pitch(
                    lowestPitch=lowestPitch, highestPitch=highestPitch)
            if(currentSymbol == '+'):
                pitch = pitch + \
                    random_note_pitch(lowestPitch=lowestPitch,
                                      highestPitch=highestPitch)
                if(pitch > highestPitch):
                    pitch = highestPitch
                elif(pitch < lowestPitch):
                    pitch = lowestPitch
            if(currentSymbol == '-'):
                pitch = pitch - \
                    random_note_pitch(lowestPitch=lowestPitch,
                                      highestPitch=highestPitch)
                if(pitch > highestPitch):
                    pitch = highestPitch
                elif(pitch < lowestPitch):
                    pitch = lowestPitch
        return mp.chord(randomChord, interval=random.choice(durations))
```

A brief explanation for Code Snippet 12 follows. The L-System's output, aptly named *output*, is iterated over in chain. For each iteration of the first loop, a new note is created. The second

loop serves to determine the information related to that note (i.e., pitch, duration, and volume) as well as determining when a new note needs to be created. That said, for every loop iteration the respective symbol is obtained, that is, *currentSymbol* and *nextSymbol*. If *currentSymbol* equals *nextSymbol* and *currentSymbol* is an “F”, it means the same note is being built, so its duration is increased randomly. If *currentSymbol* differs from *nextSymbol* and *currentSymbol* is an “F”, it means that a new note will be built in the next iteration. Like so, we need to register the information that was being built up to this point. A new note is created, and it is added to *randomChord*, which houses all instantiated notes. In the event of *currentSymbol* being a “+” or “-” symbol, the pitch of the note being currently built will shift randomly.

From a musical perspective, there is not much to be discussed about this interpretation given its random nature. Besides controlling the duration of notes with the repetition or non-repetition of the “F” symbol, which may result in a wider variety of rhythm, there is not much control of the musical aspects of this interpretation²⁵.

The reader can listen to some generated random interpretations of L-Systems, available at <https://soundcloud.com/user-62526899/sets/l-music-random-interpretations-of-l-systems>.

7.3.2.2 Scale-Based Interpretation

The implementation for the scale-based interpretation is presented in Code Snippet 13.

²⁵ One could argue that having a random interpretation on an L-System nullifies their self-similar and repetitive nature which might prove useful for music generation, given that music itself also relies on form and structure.

Code Snippet 13 - Implementation of the scale-based interpretation of an L-System

```
import random
import musicpy as mp

def scale_pitches(scaleRoot, scaleMode, octaveNumber=1, rootPitch='C4'):
    scalePitches = []
    scale = mp.scale(scaleRoot, scaleMode)
    for i in range(octaveNumber):
        pitch = i + int(rootPitch[1])
        scale = mp.scale(rf'{scaleRoot[0]}{pitch}', scaleMode)
        for note in scale.notes:
            scalePitches.append(note.degree)
    return list(dict.fromkeys(scalePitches))

def scale_parsing(output, scaleRoot, scaleMode, noteDuration=1/8, durations=[1/32, 1/16, 1/8,
1/4, 1/2], volume=100, octaveNumber=1, rootPitch='C4'):
    scaleChord = []
    scalePitches = scale_pitches(
        scaleRoot, scaleMode, octaveNumber=octaveNumber, rootPitch=rootPitch)
    for i in range(len(output)):
        currentSymbol = output[i]
        duration = noteDuration
        pitch = scalePitches[random.randint(0, len(scalePitches)-1)]
        newNote = mp.degree_to_note(pitch, duration=duration)
        for j in range(1, len(output)):
            nextSymbol = output[j]
            if(currentSymbol == nextSymbol and currentSymbol == 'F'):
                duration += noteDuration
            if(currentSymbol != nextSymbol and currentSymbol == 'F'):
                newNote = mp.degree_to_note(
                    pitch, duration=duration, volume=volume)
                scaleChord.append(newNote)
                duration = random.choice(durations)
                pitch = scalePitches[random.randint(0, len(scalePitches)-1)]
            if(currentSymbol == '+'):
                if(pitch == scalePitches[-1]):
                    pitch = scalePitches[0]
                pitch = scalePitches[scalePitches.index(pitch)+1]
            if(currentSymbol == '-'):
                if(pitch == scalePitches[0]):
                    pitch = scalePitches[scalePitches.index(pitch)-1]
                pitch = scalePitches[scalePitches.index(pitch)-1]
    return mp.chord(scaleChord, interval=noteDuration)
```

A brief explanation of Code Snippet 13 follows. Some logic remains from the random interpretation, meaning that (i) the chained loops over the L-System's output remain the same and (ii) the mechanisms for deciding when a note is created, or its duration is extended also

remain the same. Where the scale-based interpretation varies is what pitches it uses and how it decides to change them in the face of “+” and “-” symbols. The starting pitch of the first note is a randomly selected one that is contained within the scale’s note list (e.g., if the user selected the C Major scale, this method does not create a C# note). The scale’s notes list After that, the pitch of the following notes will be higher pitches within the scale if *currentSymbol* is “+” or lower pitches within the scale if *currentSymbol* is “-”. Checks were added to cover scenarios where there is a sequence of “+” or “-” symbols that exceed *scalePitches*’ size. If this happens, the note pitch will resolve to the first value in *scalePitches* if the symbol is a “+” or the last value in *scalePitches* if the symbol is a “-”.

From a musical standpoint, the scale-based interpretation will produce more structured and consonant results since it only utilizes note pitches within a given scale and within a given octave range. Another consideration for this interpretation is pitch shifting. If an L-System’s output proves varied enough in terms of lengths of sequences of “+” and “-” symbols, the musical result will prove more interesting because pitch shifts will happen in more ways (e.g., in intervals of 3rds, 4ths and 5ths instead of just being 3rds). Finally, the allowed rhythmic figures are more selective than the random interpretation. For the scale-based interpretation, thirty-two second, sixteenth, eighth, quarter and half notes are allowed. By decreasing this number, we will also obtain a more consistent rhythmic structure when compared to the random interpretation. A potential disadvantage of this approach is that there might not be a lot of development in the fragments, i.e., they will not give the listener a sense of a beginning, middle and end, but rather a continuous loop²⁶.

The reader can listen to some generated scale-based interpretations of L-Systems at <https://soundcloud.com/user-62526899/sets/l-music-scale-based>.

7.3.2.3 Polyphonic Interpretation

Two implementations were made for interpreting L-Systems musically in a polyphonic fashion. Code Snippet 14 provides the implementation that relied on just using the “F” symbol to create two voices. Code Snippet 15 provides the implementation that relied on using “F” and “G” symbols to create two voices.

²⁶ This is something that the musician or composer using L-Music would potentially have to overcome when using its generated fragments during their writing process.

Code Snippet 14 - First Implementation of a Polyphonic interpretation of an L-System, relying only on “F” symbols

```
import random
import musicpy as mp

def polyphonic_parsing(output, scaleRoot, scaleMode, noteDuration=1/8, durations=[1/32, 1/16, 1/8, 1/4, 1/2], volume=100, octaveNumber=1, rootPitch='C4'):
    voice1Chord = []
    voice2Chord = []
    scalePitches = scale_pitches(
        scaleRoot, scaleMode, octaveNumber=octaveNumber, rootPitch=rootPitch)
    for i in range(len(output)):
        currentSymbol = output[i]
        duration = noteDuration
        duration2 = noteDuration / 8
        pitch = scalePitches[random.randint(0, len(scalePitches)-1)]
        newNote = mp.degree_to_note(pitch, duration=duration)
        for j in range(1, len(output)):
            nextSymbol = output[j]
            if(currentSymbol == nextSymbol and currentSymbol == 'F'):
                duration += noteDuration
                duration2 += noteDuration / 4
            if(currentSymbol != nextSymbol and currentSymbol == 'F'):
                newNote = mp.degree_to_note(
                    pitch, duration=duration, volume=volume - 25)
                newNote2 = newNote.down(4)
                newNote2.volume = 75
                newNote2.duration = duration2
                voice1Chord.append(newNote)
                voice2Chord.append(newNote2)
                duration = random.choice(durations)
                pitch = scalePitches[random.randint(0, len(scalePitches)-1)]
            if(currentSymbol == '+'):
                if(pitch == scalePitches[-1]):
                    pitch = scalePitches[0]
                pitch = scalePitches[scalePitches.index(pitch)+1]
            if(currentSymbol == '-'):
                if(pitch == scalePitches[0]):
                    pitch = scalePitches[scalePitches.index(pitch)-1]
                pitch = scalePitches[scalePitches.index(pitch)-1]

    return [mp.chord(voice1Chord, interval=noteDuration), mp.chord(voice2Chord, interval=noteDuration)]
```

Implementation-wise, Code Snippet 14 is similar to the scale-based interpretation, with the exception that two voices are controlled, *voice1Chord* and *voice2Chord*. Its value is found when analyzing it from a musical standpoint. Essentially, in this approach, *voice2Chord*'s musical line depends on and is dictated by *voice1Chord*. This allows the establishment of

harmony between the two voices. Pitch wise, *voice2Chord* will play a major third (four semitones) below *voice1Chord*. For note duration, *voice2Chord*'s notes will be an eighth note shorter than *voice1Chord*'s notes.

The reader can listen to some generated polyphonic interpretations on L-Systems, using the aforementioned approach, at <https://soundcloud.com/user-62526899/sets/l-music-polyphonic-interpretations-on-l-systems>.

The implementation using the “G” symbols is displayed in Code Snippet 15.

Code Snippet 15 - Polyphonic Implementation using "F" and "G" symbols

```
def polyphonic_parsing(output, scaleRoot, scaleMode, noteDuration=1/8, durations=[1/32, 1/16,
1/8, 1/4, 1/2], volume=100, octaveNumber=1, rootPitch='C4'):
    voice1Chord = []
    voice2Chord = []
    scalePitches = scale_pitches(
        scaleRoot, scaleMode, octaveNumber=octaveNumber, rootPitch=rootPitch)
    for i in range(len(output)):
        currentSymbol = output[i]
        duration = noteDuration
        duration2 = noteDuration / 8
        pitch = scalePitches[random.randint(0, len(scalePitches)-1)]
        pitch2 = pitch + 12
        newNote = mp.degree_to_note(pitch, duration=duration)
        for j in range(1, len(output)):
            nextSymbol = output[j]
            if(currentSymbol == nextSymbol and currentSymbol == 'G'):
                duration2 += noteDuration / random.choice([2, 4, 8])
            if(currentSymbol != nextSymbol and currentSymbol == 'G'):
                newNote2 = mp.degree_to_note(
                    pitch2, duration=duration, volume=volume - 25)
                newNote2.volume = 75
                voice2Chord.append(newNote2)
            if(currentSymbol == nextSymbol and currentSymbol == 'F'):
                duration += noteDuration
            if(currentSymbol != nextSymbol and currentSymbol == 'F'):
                newNote = mp.degree_to_note(
                    pitch, duration=duration, volume=volume - 25)
                voice1Chord.append(newNote)
                duration = random.choice(durations)
                pitch = scalePitches[random.randint(0, len(scalePitches)-1)]
            if(currentSymbol == '+'):
                if(pitch == scalePitches[-1]):
                    pitch = scalePitches[0]
                    pitch2 = pitch + 4
                pitch = scalePitches[scalePitches.index(pitch)+1]
                pitch2 = pitch + 12
            if(currentSymbol == '-'):
                if(pitch == scalePitches[0]):
                    pitch = scalePitches[scalePitches.index(pitch)-1]
                    pitch2 = pitch + 4
                pitch = scalePitches[scalePitches.index(pitch)-1]
                pitch2 = pitch - 12
        return [mp.chord(voice1Chord, interval=noteDuration), mp.chord(voice2Chord,
interval=noteDuration)]
```

To consider “G” symbols in Code Snippet 15, two conditions, similar to those of “F” symbols to know when to build or play a note that is associated with the “G” symbol depending on the value of *currentSymbol*. The *pitch2* variable is introduced here, which starts its value an octave (12 semitones) up from *pitch*. Whenever “+” or “-” symbols are encountered, *pitch2* will alter

its value based on *pitch* as well. If a “+” symbol appears and *pitch* needs to be reset to the first pitch in *scalePitches*, *pitch2* will be a major third (4 semitones) up from *pitch*. Otherwise, it will be an octave (12 semitones) up from *pitch*. If a “-” symbol appears and *pitch* needs to be reset to the last pitch in *scalePitches*, *pitch2* will be a major third up from *pitch*. Otherwise, it will be an octave down from *pitch*.

From a musical point-of-view, the voice represented by “G” still follows the lead of the voice represented by “F”, albeit slightly differently. Rhythm-wise, the durations of notes created by “G” symbols are not dependent on the durations of notes created by “F” symbols. However, it is ensured that the rhythm of these notes is kept somewhat consistent, only with slight variations in their duration. This succeeds in having a structured rhythm form. The pitch of notes created by “G” symbols have a dependency on the pitch of notes created by “F” symbols., varying either a major third up or down, or an octave up or down from them. This will keep the voice created by “G” symbols with less variation in pitch, making it an accompaniment voice for the notes created by “F” symbols.

The reader can listen to some examples generated by this approach at <https://soundcloud.com/user-62526899/sets/l-music-polyphonic-interpretation-on-l-systems-using-2-symbols>.

7.3.3 Remarks on the generated musical fragments

This section aims at providing general comments on some of the properties of the generated musical fragments presented in 7.3.2.

First, it is necessary to consider the grounds on which these fragments were generated as well as the limitations that are imposed by L-Systems, specifically:

- L-Systems are nonadaptive (see 2.2), meaning that over time they will not evolve into new results. Their self-similar properties reinforces the latter, which means that producing longer fragments of music that have a continuous sense of development in melody, rhythm or dynamics proves to be a difficult challenge. However, given that *L-Music* is an assistive system, the main interest is that it is capable of providing small ideas for musicians and composers to use in their own works. This is discussed further in section 9.2, as this can be considered a limitation that L-Systems have in assisted and autonomous music composition (a composer may want longer excerpts of musical ideas for their ideas).
- The implementation of the various musical interpretations presented in 7.3.2 relied on a small set of reserved symbols. Furthermore, due to time constraints, only deterministic context-free L-Systems were used. From a technical standpoint, *L-Music* is apt to support other types. By applying the latter and adding more symbols with specific interpretations (e.g., for handling volume changes or rhythm changes, or using brackets to manage the state of the musical interpretation), the results will

prove even more varied than the ones that were obtained. This is discussed further in section 9.2.

Since the total number of generated fragments is moderately sized (18), the focus will be on a specific fragment per implemented musical interpretation. Therefore, the following excerpts are considered:

- For the random interpretation, the following excerpt was selected <https://soundcloud.com/user-62526899/l-music-random-1>.
- For the scale-based interpretation, the following excerpt was selected <https://soundcloud.com/user-62526899/l-music-scale-based-5>.
- For the first approach on polyphonic interpretation, the following excerpt was selected <https://soundcloud.com/user-62526899/l-music-polyphonic>.
- For the second approach on polyphon interpretation, the following excerpt was selected <https://soundcloud.com/user-62526899/l-music-polyphonic-3>.

For each generated excerpt, the L-System and its number of iterations will be provided and a brief discussion on the musical quality of the former. Considering the subjective nature of music, the discussion on musical quality is strongly tied to the author's own subjective perceptions and opinions on music and music composition.

Random Interpretation on Koch Curve

The Koch Curve can be represented by the following L-System:

An alphabet consisting of the symbols "F", "+" and "-".

An axiom consisting of the symbol "F".

A production rule which states that predecessor "F" turns into successor "F+F--F+F".

Number of iterations: 1.

Bearing in mind that the random interpretation relies on the L-System's output to a minimum extent (i.e., almost all properties of the fragment were determined by chance), the only thing to note in this fragment is that the variation of rhythm and volume along the sequence of notes makes the fragment more interesting and enjoyable to hear.

Scale-Based Interpretation on Peano Curve

The Peano Curve can be represented by the following L-System:

An alphabet consisting of the symbols "X", "Y", "F", "+" and "-".

An axiom consisting of the symbol "X".

A production rule set consisting of two rules:

Predecessor "X" turns into successor "XFYFX+F+YFXFY-F-XFYFX".

Predecessor “Y” turns into successor “YFXFY-F-XFYFX+F+YFXFY”.

Number of iterations: 1.

The generated melody is interesting, albeit simple. It has a structured form and contour over its total duration. Pitch shifts do not happen in large intervals, with the exception of one or two notes. Due to the selected MIDI Instrument’s properties (guitar), some notes are played in *staccato*²⁷, which compensates the lack of rhythmic variation.

Polyphonic Interpretation (First Approach) on Dragon Curve

The Dragon Curve can be represented by the following L-System:

An alphabet consisting of the symbols “F”, “X”, “Y”, “+” and “-”.

An axiom consisting of the word “FX”.

A production rule set consisting of two rules:

Predecessor “X” turns into successor “X+YF+”.

Predecessor “Y” turns into predecessor “-FX-Y”.

Number of iterations: 5.

The listening experience is much richer considering that there are now two voices that the listener can pay attention to. The generated fragment is quite interesting to hear, and sonically it is a bit intense and somewhat chaotic, but with a defined structure (the latter being propelled by the self-similarity found in L-Systems). There is a good amount of dissonance happening between the stringed instruments (in this case cellos and violins) throughout the whole piece which motivates the listener to be focused on it throughout its whole length. If the listener pays close attention to the cellos, they will notice it mostly plays short lived notes and establishes a somewhat stable rhythmic pattern for the entirety of the excerpt, while sometimes also performing a callback and response trade with the violins, which play a sometimes-clashing melody against the cellos. This fragment could potentially be used as a marking point in a piece, where a transition into an overwhelming and chaotic situation occurs from a more consonant and peaceful state of mind.

Polyphonic Interpretation (Second Approach) on Sierpinski Triangle

The Sierpinski Triangle can be represented by the following L-System:

An alphabet consisting of the symbols “F”, “G”, “+” and “-”.

An axiom consisting of the word “F-G-G”.

²⁷ *Staccato* means “to play a note distinctly detached and separated (i.e., silence) from the notes that follow it”.

A production rule set consisting of two rules:

Predecessor “F” turns into successor “F-G+F+G-F”.

Predecessor “G” turns into successor “GG”.

Number of iterations: 2.

Similarly, to the first approach on polyphonic interpretation, the listening experience is more vivid due to the existence of two instruments being played. Where this approach differs is how much more consonant and harmonious it is than the first approach. Additionally, the L-System itself also provides a more cohesive and repetitive structure to the fragment. There are two clear lines. One plays a smaller set of notes repetitively (represented by a MIDI xylophone) while the other plays a melodic line that consists of sequential ascents in descents of note pitches (represented by MIDI tubular bells) and is repeated throughout the total length of the fragment (the self-similar nature of L-Systems is present once more). In other words, the concept of having a voice that is accompanying the other is verified here again. Sonically, the fragment is light-hearted and metallic, due to the scale that was used (C Major scale) and the selected instruments). While it is repetitive, the melody produced by the tubular bells keeps the listener engaged in a passive manner. It could potentially be utilized as a background music for other forms of media (e.g., video games or a television show) or inserted in a piece to establish its main ambience while other instruments would come into play. Finally, near the end of the fragment the listener will stop hearing the tubular bells. This indicates that there could be improvements to be done in the implementation with regards to controlling the total duration of each voice (ideally, both would end at the same time or near each other).

7.4 Chapter Summary

This chapter documented the implementation of the *L-Music* prototype. Divided into three sections, each demonstrated the work done for the core modules of the application: the user interface module; the L-System module and the Musical Parser module.

8 Solution Assessment

This chapter encompasses the assessment and experiments done for the developed solution, presented in chapter 7. Its outline follows. Section 8.1 details the components that were involved in assessing the developed solution. Section 8.2 explains what tests were done to evaluate the components laid out in 8.1. Finally, section 8.3 discusses the obtained results from the performed tests.

8.1 Assessment Components

Considering that *L-Music* is an assisted music composition tool, and that L-Systems are a concept that is outside of the musical realm, the following components were identified as relevant for testing the quality of the developed application:

- Usability. This ranges from the factors identified in 4.4.2.2 as well as the ones that were considered in the design of the user interface and experience (see 6.2).
- How knowledge on L-Systems, or a lack thereof from the user, impacts the experience when interacting with *L-Music*.
- How knowledge on music and music theory, or a lack thereof, from the user, impacts the experience when interacting with *L-Music*.
- Whether or not *L-Music* serve its goal, which is to suggest ideas for composing music.
- Whether the quality of the musical fragments generated by *L-Music* are musically relevant and pleasant to the human ear.

8.2 Tests

In order to assess the components listed in 8.1, surveys such as the System Usability Scale (SUS) [153] and the Discrete Emotions Questionnaire (DEQ) [36] could be employed given a large enough number of participants to draw statistically significant data about the developed prototype. Due to time constraints this was not feasible and as such only qualitative data was

collected from a total of two participants. Notwithstanding, an attempt was made to collect the information, albeit in an informal manner. The two participants shall be identified from this point onwards as “Participant A” and “Participant B”.

Two tests were done with each participant. The first test involved a makeshift usability session, with the author acting as the guide for each participant. The participants were asked to describe their thought process and reasoning as they interacted with the system freely. The author would clarify any doubts they could have about the system if needed. All functionalities of *L-Music* were tested by each participant. An important note to make about this test is that the author, prior to the session’s beginning, explained to one of the participants what L-Systems were and how they worked. To the other participant, no explanation on L-Systems was given. Furthermore, some questions were made to both participants to gather some information on their musical capabilities and experience with music related software. This information is listed in Table 4. In the second test, the participants were asked to participate in a hearing session. Both, unknowingly, listened to previously generated musical excerpts from *L-Music* and were asked to state whether the application had generated them or if it was a composition from a human. Additionally, the author asked the participants to comment on the listening experience they had felt while listening to the fragments. These excerpts are the ones found in section 7.3.2.

Table 4 - Collected data from participants prior to beginning test session

| Question | Answers from Participant A | Answers from Participant B |
|---|--|--|
| <i>“What would you say is your proficiency and knowledge on music theory and music composition?”</i> | Superficial knowledge on music theory. No knowledge on music composition. | Intermediate knowledge on both subjects. |
| <i>“Do you play a musical instrument? If yes, for how long have you been playing it and how proficient do you consider yourself to be?”</i> | Electric guitar intermittently for seven years. Considers having intermediate proficiency with the instrument. | Alto saxophone regularly for eleven years. Considers having intermediate proficiency with the instrument. |
| <i>“Are you familiar with music composition/production software such as DAWs or notation software?”</i> | No familiarity with these applications. | Some familiarity with these programs. |
| <i>“Do you know anything about Lindenmayer Systems?”</i> | No knowledge on L-Systems. (L-Systems were explained to this participant prior to the test session beginning) | No knowledge on L-Systems. (L-Systems were not explained to this participant prior to the test session beginning). |

8.3 Results

Considering that the tests mentioned in 8.2 were carried out in an informal fashion, results were collected the same way. During both tests, the author registered the participants feedback. The tests were carried out separately with each participant.

8.3.1 Usability Session

Regarding the system's usability, Participant A's feedback was the following:

- The native dialog for selecting a SoundFont File (with extension *.sf2*) shows all files that the device has (i.e., the system does not filter by file extension *.sf2*) (see 7.1.1). This gives the user a chance to introduce an invalid file to the system. If this happens, the system does not crash, but it does not warn the user that they loaded an invalid file type.
- The system should provide a section that explains what L-Systems are and how they work. Otherwise, it is confusing to see an "L-Systems" list after loading a SoundFont file²⁸.
- The system allows the user to hear multiple instrument samples at once (see 7.1.2.1). Participant's A expectation was that the system would pause the previous sample and play the next one as to avoid a cacophony of instruments. This feedback also applies to the scale audio samples (see 7.1.2.4).
- Some text was hard to read, particularly the error messages presented by the text fields when creating custom L-Systems (see 7.1.4). Participant A attributed this difficulty to the text's font size.
- A lack of keyboard shortcuts or other accelerators proves frustrating after the user is familiarized with the system and wants to achieve a higher productivity rate.
- The system is not error prone, and as such Participant A faced system freezes or crashes without context²⁹.
- Some musical fragments took a considerable amount of time to be generated (e.g., with a high number of L-System iterations). As such, Participant A was expecting that the system provided a way to cancel the generation to not get stuck. Additionally, Participant A expressed that a discrete progress indicator, rather than the implemented continuous progress indicator (see 7.1.2.4), would aid the user in understanding if the system is almost finished or not with generating the musical fragment.

²⁸ This feedback was provided by Participant A considering a scenario where they had not been explained what L-Systems were.

²⁹ This was somewhat expected given that Flutter for desktop platforms is still in a beta phase of development [154].

- Some list views were quite extensive. Participant A expressed that a search bar and a filter system for such lists (e.g., the instrument list or the scale list) would be beneficial to get to the desired item more quickly.
- After creating multiple musical excerpts, it is tiring to switch between back and forth using the skip and previous buttons (see 7.1.2.5). Participant A expressed that having a playlist with all of them next to these components would aid in a quicker switch between them.
- Participant A expressed that having opaque list views was not a proper visual indicator for the musical parser parameters (see 7.1.3). Additionally, the participant thought that those parameters were still being considered due to some items in those list views remaining in a selected state.
- The Musical Parser List's items should come in an expanded state by default (see 7.1.2.3). Participant A gave this feedback since they wanted to know what a given Musical Parser would do prior to selecting it and start editing its corresponding musical parameters.
- Participant A expressed a desire to save their current configuration (i.e., selected Instrument, selected L-System, selected Musical Parser and parameters) into a file that could be later loaded onto *L-Music*. Additionally, the participant also conveyed that having an accompanying graphical visualization of the L-Systems would be a pleasant addition to the application.

Regarding the system's usability, Participant B's feedback was the following:

- The lack of a splash screen affected Participant's B introduction to the system. There was a lack of a call to attention on what the participant needed to do.
- Upon loading a SoundFont file, Participant B felt slightly overwhelmed by the amount of information that was displayed (see 7.1.1).
- Since Participant B was not explained L-Systems prior to the session beginning, the participant thought that the instrument sample served as the basis for any music being generated in the application. For the participant, L-Systems would play the role of transforming that sample into something new based on the selected musical parser.
- Upon selecting the polyphonic musical parser, Participant B did not understand that it was required to select a second instrument from the instruments list. To this effect, the participant suggested that the musical parser list item should have a visual aid to convey the information to the user (e.g., an icon).
- Participant B also faced system errors, namely the generation of musical fragments would sometimes fail. Since no error message was shown, it was not clear to the participant that the generation had not been successfully completed.
- Participant B informed that having control over the duration of the generated fragments as well as its meter would prove very useful to aid in writing a musical piece. The latter, as well as other concepts in music theory, could be inserted into an "advanced options" view, since some users might not understand the concepts of meter

- From a musical perspective, Participant B suggested that the scale list be organized in a different manner, where one could first select the root note of the scale and then its mode rather than having a big list with all possible scales.
- Given that Participant B did not know what L-Systems were, the participant provided a suggestion to include a tooltip component next to the list that could offer a brief explanation to the user of how L-Systems work or at least how they impacted the generated musical fragments.

Both participants gave positive feedback with regards to the visual hierarchy and flow of *L-Music*, stating that it was simple and effective to get some output out of the system. Creating a custom L-System was also perceived to be a simple and concise experience, provided that the user knew how L-Systems worked. Without this knowledge, the user might have a harder time understanding how to create one, even with textual hints provided by the system.

8.3.2 Hearing Session

The hearing session was performed shortly after each usability session with each participant. The participants had not listened to any generated fragments from *L-Music* besides the ones they had created themselves during their respective usability sessions. The excerpts that were used were all generated by *L-Music* and are the ones presented in 7.3.2. For readability purposes, Table 5 maps each excerpt to a name.

Table 5 - Generated Excerpts from *L-Music* that were used for a hearing session

| Generated Excerpt Identifier | Resource link |
|------------------------------|---|
| R0 | https://soundcloud.com/user-62526899/l-music-random-interpretation |
| R1 | https://soundcloud.com/user-62526899/l-music-random-1 |
| R2 | https://soundcloud.com/user-62526899/l-music-random-interpretation-on-peano-curve |
| R3 | https://soundcloud.com/user-62526899/l-music-random-interpretation-on-koch-island-two-iterations |
| S0 | https://soundcloud.com/user-62526899/l-music-scale-based-6 |
| S1 | https://soundcloud.com/user-62526899/l-music-scale-based-1 |
| S2 | https://soundcloud.com/user-62526899/l-music-scale-based-2 |
| S3 | https://soundcloud.com/user-62526899/l-music-scale-based-3 |
| S4 | https://soundcloud.com/user-62526899/l-music-scale-based-4 |
| S5 | https://soundcloud.com/user-62526899/l-music-scale-based-5 |
| PF0 | https://soundcloud.com/user-62526899/l-music-polyphonic-interpretation-on-koch-curve-5-iterations-in-e-harmonic-minor |
| PF1 | https://soundcloud.com/user-62526899/l-music-polyphonic |
| PF2 | https://soundcloud.com/user-62526899/l-music-polyphonic-1 |
| PFG0 | https://soundcloud.com/user-62526899/custom-lsystem-poly-fg-dmaj |
| PFG1 | https://soundcloud.com/user-62526899/l-music-polyphonic-2 |
| PFG2 | https://soundcloud.com/user-62526899/l-music-polyphonic-3 |

An explanation of the identifiers in Table 5 follows. Identifiers starting with “R” are excerpts that were generated using the random interpretation (see 7.3.2.1). Identifiers starting with “S” are excerpts that were generated using the scale-based interpretation (see 7.3.2.2). Identifiers starting with “PF” are excerpts that were generated using the first polyphonic approach and identifiers starting with “PFG” are excerpts that were generated using the second polyphonic approach (see 7.3.2.3).

Each participant would hear an excerpt and state if it was composed by a human composer or by *L-Music* and provide a reasoning as to why. The results from the hearing session are in Table 6.

Table 6 - Results from hearing session to determine if generated excerpts from *L-Music* could be judged as human composed

| Generated Excerpt Identifier | Participant A's Responses | Participant B's Responses |
|------------------------------|---------------------------|---------------------------|
| R0 | <i>L-Music</i> | <i>L-Music</i> |
| R1 | Human | Human |
| R2 | <i>L-Music</i> | <i>L-Music</i> |
| R3 | <i>L-Music</i> | <i>L-Music</i> |
| S0 | Human | Human |
| S1 | Human | <i>L-Music</i> |
| S2 | <i>L-Music</i> | <i>L-Music</i> |
| S3 | Human | Human |
| S4 | <i>L-Music</i> | <i>L-Music</i> |
| S5 | Human | Human |
| PF0 | <i>L-Music</i> | <i>L-Music</i> |
| PF1 | Human | <i>L-Music</i> |
| PF2 | Human | <i>L-Music</i> |
| PFG0 | <i>L-Music</i> | <i>L-Music</i> |
| PFG1 | Human | Human |
| PFG2 | Human | <i>L-Music</i> |

With regards to randomly generated excerpts, only R1 was guessed as being human composed by both participants. Both participants justified their choice mainly due to the note sequence being played, as well as its volume variation. The other ones were quickly dismissed due to their overall randomness.

With regards to scale-based generated excerpts, both participants identified S0, S3 and S5 as being human composed. Additionally, Participant A also attributed S1 to being composed by a human. The justifications from both participants once again revolved around the variation in rhythm but also the melodic contour of each excerpt. Repetitiveness in structure and a lack of continuous development were both reasons given by the participants to identify the other scale-based excerpts as being written by *L-Music*.

With regards to polyphonic generated excerpts using the first approach, only Participant A identified excerpts as being composed by human, specifically, PF1 and PF2. Nevertheless, it should be noted that Participant B's choice on PF1 was due to the participant's opinion, based on their musical knowledge, that the rhythm structure found on PF1 was something that a

human composer would not typically think of³⁰. Besides this case, the other excerpts were identified as being human composed or composed by *L-Music* for the same reasons provided for the scale-based excerpts.

With regards to polyphonic generated excerpts using the second approach, both participants identified PFG1 as being human composed. Participant A also identified PFG2 as being human composed. The reasons for these identifications were the same as the previously mentioned excerpts.

Both participants expressed that most excerpts were sonically interesting and pleasing to listen to, with excerpts R0, R1, S3, S5, PF1, PFG1 and PFG2 standing out. Furthermore, both also expressed that the generated excerpts would help musicians and composers in their creative process, since the excerpts were musically interesting. Regarding PF1 specifically, because of its varying rhythms and clashing melodies, both participants expressed feeling a sense of an overwhelming chaos. Participant A stated the following about PF1:

“It feels like I am watching a movie about a character who’s had their life turned upside down, and now they are trying to figure out what to do and how to fix things.”

Participant B described PF1 as the following:

“It feels like someone who is in an office, working, and people just keep stacking papers on their desk nonstop, and at some point, they start to feel overwhelmed by it all.”

It is the author’s personal opinion that having both participants react in such similar ways to hearing the same musical excerpt is quite interesting, and that it is something that happens quite frequently with the art form of music.

8.4 Chapter Summary

This chapter served to document a qualitative assessment of the developed prototype for an assisted music composition system named *L-Music*. First, the relevant components for the assessment were identified. After, a description of the test conditions as well as the total number of participants and what the tests involved were presented. Finally, an overview on the obtained results from a usability session and a hearing session with the participants was reported.

³⁰ This alludes to some topics discussed in 1.2.2 regarding computational creativity.

9 Conclusions

This chapter contains the conclusions of the work that was done for the problem presented in chapter 1. Its outline follows. In it, a discussion about the goals that were achieved are discussed first, in section. Future work suggestions are laid out after. Lastly, the reader can confer some of the author's personal remarks on the developed work.

9.1 Achieved Goals

To understand what goals were accomplished, we set out to answer the three questions presented in section 1.3 as well as review the state of the use cases presented in section 4.4.1 and the state of non-functional requirements presented in section 4.4.2.

Can algorithmic music be of assistance to musicians or composers and augment their creativity?

We believe that the answer is yes. From the reviewed state of the art, algorithmic music is ever-growing. Furthermore, the reviewed works and studies already showcased that algorithmic music can indeed support (and in some cases, even replace) musicians and composers (see 2.4.5, for example).

What are the best techniques that algorithmic music should rely on?

The answer to this is inconclusive as we were not able to compare the developed solution, which resorted to L-Systems, against other approaches, such as Deep Learning or genetic algorithms (and even if the comparison were done, it would still prove hard to reach to a definitive answer). From the reviewed state of the art, all approaches have potential and can be used to generate interesting results. Additionally, we believe that the developed work further demonstrates the potential that L-Systems possess for assisted music composition systems. Nevertheless, the latter lacks the support and validation from a proper assessment, unlike the informal one that was performed.

What interactions will users expect from such a system?

Users will expect a system that can model and abstract musical concepts into terms they are familiar with. Moreover, the system should provide several degrees of complexity to accommodate for multiple types of users. Lastly, because it is a system intended to augment their user's creativity, the system should allow as much customization as possible with regards to, in this case, composing musical excerpts.

Table 7 presents the state of the Use Cases defined in section 4.4.1.

Table 7 - State of defined Use Cases at the end of the project

| Use Case | State | Missing |
|----------|-----------------------|---|
| UC01 | Implemented | Tests |
| UC02 | Implemented | Tests |
| UC03 | Implemented | Do not allow multiple samples to be played at the same time; Tests |
| UC04 | Implemented | Tests |
| UC05 | Implemented | Tests |
| UC06 | Implemented | Tests |
| UC07 | Implemented | Tests |
| UC08 | Implemented | Tests |
| UC09 | Partially Implemented | Integration with other L-System types (context-sensitive, stochastic and table); Musical Interpretations missing (polyrhythmic, chord progression, classy and jazzy); Tests |
| UC10 | Implemented | Tests |
| UC11 | Implemented | Tests |
| UC12 | Implemented | Tests |
| UC13 | Partially Implemented | Integration with other L-System types (context-sensitive, stochastic and table); Tests |

All defined use cases were implemented, with the exception of UC09 and UC13, where integration with other L-Systems were missing. Additionally, no tests (e.g., unit tests and integration tests) were implemented during the implementation of the *L-Music* prototype, which hinders its overall quality.

Table 8 presents the state of the non-functional requirements defined in section 4.4.2.

Table 8 - State of defined non-functional requirements at the end of the project

| Non-Functional Requirements (FURPS+) | Requirement fulfilled? | Missing |
|--------------------------------------|--|---|
| Functionality | | |
| <i>Error Handling</i> | Partially | Prototype sometimes crashes and freezes. |
| <i>Portability</i> | Partially | Prototype was not tested on a macOS machine. |
| Usability | | |
| <i>Aesthetics</i> | Yes, but should be improved iteratively. | - |
| <i>Learnability</i> | Yes, but should be improved iteratively. | - |
| <i>Memorability</i> | Yes, but should be improved iteratively. | - |
| <i>Errors</i> | Partially | Prototype faced random crashes and freezes. Musical excerpt generation would also fail periodically. |
| <i>Satisfaction</i> | Yes, but should be improved iteratively. | - |
| <i>Efficiency</i> | Partially | Audio file generation would take too much time for longer excerpt lengths. |
| Reliability | | |
| - | - | - |
| Performance | | |
| <i>Efficiency</i> | Partially | Recursion was not applied to L-System implementation |
| Supportability | | |
| - | - | - |
| Plus | | |
| <i>Implementation</i> | Yes | - |

Most non-functional requirements were fulfilled partially. The reader should also note that the fulfillments of Usability requirements are merely indicative, as the testing that was reported in 8.2 was done in an informal manner and did not gather any statistically significant data.

9.2 Limitations and Future Work

This section provides limitations that the author found during the development of this project as well as noteworthy topics for future work.

Limitations

Time constraints were a direct limitation on the developed work. Some functional and non-functional requirements were not complete due to it. Additionally, this also affected the performed assessment of the solution, which did not collect any statistically significant data.

As stated in section 7.3.3, L-Systems are nonadaptive, meaning that they do not change themselves overtime to change their output. Since music composition is very attached to the composer's mindset and ideas on music, if an assisted music composition system cannot adapt itself to the former, it would prove to be a not very useful tool³¹. With regards to L-Systems specifically, the lack of an adaptive nature can be compensated by using various types of L-Systems. More complex ones, such as environmentally sensitive or open world L-Systems, which react with information that is outside of the L-System, may prove useful for this (e.g., two L-Systems that represent an instrument and communicate between each other as they grow to ensure consonance and harmony). Additionally, increasing the extensibility of the musical interpretations on L-Systems to the musician or composer is another way to tackle the issue. By letting the former define how each symbol from an L-System's alphabet be interpreted musically, the composer is essentially creating his own musical grammar with L-Systems.

Future Work

The following items are suggestions for future work to progress with the development of the *L-Music* prototype:

- Integrate all L-System types with musical interpretations. Specifically, context-sensitive, stochastic and table systems (see section 7.3.3).
- Implement missing musical interpretations. These were, specifically, the chord progression, the polyrhythmic, the classy and the jazzy interpretations (see section 7.3.2).
- Implement new types of L-Systems. These can be extensions on the existing ones (e.g., a stochastic context-sensitive L-System) or new ones, such as hierarchical L-Systems.
- Implement more musical interpretations. These are directly related to the implementer's knowledge of the musical domain. One could, for example, implement an interpretation based on playing in note intervals (e.g., generating melodies only consisting of octaves and perfect fifths).

³¹ This is debatable, as an assisted music composition system that does not follow a composer's mindset might break his status-quo and potentially suggest new ideas that they had not considered yet, furthering their creative potential.

- Interpret more L-System symbols musically. In the developed prototype, only a total of four symbols were reserved for musical interpretation. That said, the brackets symbols, which are widely used in graphical interpretations of L-Systems to manage state, were not explored for example.
- Extend the definition of musical meaning in symbols to the composer. By doing so, we are allowing the composer to define their intended musical meaning for each symbol in an L-System's alphabet, thereby defining a musical grammar via an L-System.
- Build upon the set of developed functionalities with the feedback described in 8.3.1.
- Consider porting *L-Music* to a framework that supports VSTi plugin formats. This is as to enable musicians or composers to load *L-Music* onto their main digital tool, which is typically notation software or a DAW.
- Perform appropriate testing, using standard surveys to measure system usability, as well as custom surveys to assess whether or not *L-Music* augments composers' creativities in order to gather statistically significant data to further improve the developed prototype.
- Iterate over the implemented UI/UX design to accommodate several types of users (e.g., beginner, intermediate and advanced users).
- Add more musical parameters to be editable by the user, such as time signature or dynamic range.

9.3 Personal Remarks

I do believe that the presented developed work, albeit having flaws and gaps such as not having a formal statistically significant assessment, has contributed towards assisted music composition system research. Moreover, I think that there is still a potential in L-Systems for generating music by passing the musical significance in symbols to the hands of the musician or composer. By doing so, the creative possibilities will prove very interesting, because in essence, the user will be writing music in a different way, via musical grammars represented by L-Systems.

I also believe it to be important that research in this topic does not stop. Music history has shown that new instruments appear, new ways of thinking about how to create music appear, and both assistive and autonomous composition tools will, sooner or later, make a big impact in how we approach writing music.

Lastly, I hope my work has showcased that L-Systems can be exclusively used for generating musical ideas. From the performed review on the state of the art, it is my opinion that they are being somewhat forgotten in the world of algorithmic music, in place of trendier approaches such as neural networks. While there should be research happening for every methodology, and I will not deny the astounding results that other approaches have brought up to the table, my stance is that L-Systems require further research for a deeper analysis on their true potential.

References

- [1] K. M. Higgins, *The Music Between Us: Is Music a Universal Language?* University of Chicago Press, 2012.
- [2] R. Sabin, *Punk Rock: So What?: The Cultural Legacy of Punk*. Routledge, 2002.
- [3] A. Cohen, B. Bailey, and T. Nilsson, "The Importance of Music To Seniors," *Psychomusicology*, pp. 89–102, 2002.
- [4] L. M. Levinowitz, "The Importance of Music in Early Childhood," *Music Together*, 1998. <https://www.musictogether.com/about/research/research-based-program/importance-of-music-in-early-childhood> (accessed Dec. 06, 2020).
- [5] A. C. North, D. J. Hargreaves, and S. A. O'Neill, "The importance of music to adolescents," *British Journal of Educational Psychology*, vol. 70, no. 2, pp. 255–272, Jun. 2000, doi: 10.1348/000709900158083.
- [6] E. Schellenberg, "Cognitive Performance After Listening to Music: A Review of the Mozart Effect," in *Music, health and wellbeing*, 2012, pp. 324–338. doi: 10.1093/acprof:oso/9780199586974.003.0022.
- [7] S. Davies, "On Defining Music," *Monist*, vol. 95, no. 4, pp. 535–555, Oct. 2012, doi: 10.5840/monist201295427.
- [8] P. Alperson, *What Is Music?: An Introduction to the Philosophy of Music*. Penn State Press, 2010.
- [9] J. P. A. BURKHOLDER, D. J. Grout, C. V. Palisca, and V. Palisca, *A History of Western Music*. W.W. Norton, 1996.
- [10] M. Kaliakatsos-Papakostas, A. Floros, and M. N. Vrahatis, "Artificial intelligence methods for music generation: a review and future perspectives," in *Nature-Inspired Computation and Swarm Intelligence*, Elsevier, 2020, pp. 217–245. [Online]. Available: <https://doi.org/10.1016/B978-0-12-819714-1.00024-5>
- [11] S. A. Hedges, "DICE MUSIC IN THE EIGHTEENTH CENTURY," *Music Lett*, vol. 59, no. 2, pp. 180–187, Apr. 1978, doi: 10.1093/ml/59.2.180.
- [12] C. Saitis, "Computer-Assisted Composition," p. 3, 2010.
- [13] T. Funk, "A Musical Suite Composed by an Electronic Brain: Reexamining the Illiac Suite and the Legacy of Lejaren A. Hiller Jr.," *Leonardo Music Journal*, vol. 28, pp. 19–24, Dec. 2018, doi: 10.1162/lmj_a_01037.

- [14] L. Hiller and L. M. (Leonard M. Isaacson, *Experimental music; composition with an electronic computer*. New York, McGraw-Hill, 1959. Accessed: Dec. 06, 2020. [Online]. Available: <http://archive.org/details/experimentalmusi00hill>
- [15] S. Manousakis, "Musical L-Systems," The Royal Conservatory, The Hague, 2006.
- [16] M. A. Runco and G. J. Jaeger, "The Standard Definition of Creativity," *Creativity Research Journal*, vol. 24, no. 1, pp. 92–96, 2012, doi: 10.1080/10400419.2012.650092.
- [17] M. A. Bogden, "Creativity and artificial intelligence," *Artificial Intelligence*, vol. 103, no. 1–2, pp. 347–356, 1998, doi: [https://doi.org/10.1016/S0004-3702\(98\)00055-1](https://doi.org/10.1016/S0004-3702(98)00055-1).
- [18] G. Fernand and S. Giovanni, "How Artificial Intelligence Can Help Us Understand Human Creativity," *Frontiers in Psychology*, vol. 10, p. 1401, 2019, doi: 10.3389/fpsyg.2019.01401.
- [19] M. du Satoy, "Can AI ever be truly creative?," *New Scientist*, May 11, 2019.
- [20] S. D. Kelly, "A philosopher argues that an AI can't be an artist," 2019. Accessed: May 12, 2020. [Online]. Available: <https://www.technologyreview.com/2019/02/21/239489/a-philosopher-argues-that-an-ai-can-never-be-an-artist/>
- [21] C. McKay, "The Bach reception in the 18th and 19th centuries," *Canada, University of Guelph*, 1999.
- [22] S. D. Holcomb, W. K. Porter, S. V. Ault, G. Mao, and J. Wang, "Overview on DeepMind and Its AlphaGo Zero AI," in *Proceedings of the 2018 International Conference on Big Data and Education*, 2018, pp. 67–71. doi: 10.1145/3206157.3206174.
- [23] R. Chamberlain, C. R. Mullin, B. Scheerlinck, and J. Wagemans, "Putting the Art in Artificial: Aesthetic Responses to Computer-Generated Art," *Psychology of Aesthetics Creativity and the Arts*, vol. 12, no. 2, 2017, doi: 10.1037/aca0000136.
- [24] S. Bringsjord, P. Bello, and D. Ferrucci, "Creativity, the Turing Test, and the (Better) Lovelace Test," *Minds and Machines*, vol. 11, no. 1, pp. 3–27, Feb. 2001, doi: 10.1023/A:1011206622741.
- [25] J. McCormack, T. Gifford, and P. Hutchings, "Autonomy, Authenticity, Authorship and Intention in Computer Generated Art," in *Computational Intelligence in Music, Sound, Art and Design*, Cham, 2019, pp. 35–50.
- [26] G. Watson, "Resistance to Change," *American Behavioral Scientist*, vol. 14, no. 5, pp. 745–766, May 1971, doi: 10.1177/000276427101400507.
- [27] M. Pardo-del-Val and C. Martinez-Fuentes, "Resistance to change: A literature review and empirical study," *Management Decision*, vol. 41, pp. 148–155, Jan. 2003.

- [28] T. Pinch and F. Trocco, *Analog Days: The Invention and Impact of the Moog synthesizer*. The Harvard University Press, 2002.
- [29] S. Cohen, *Folk Devils and Moral Panics: The Creation of the Mods and Rockers*, 7th ed. New York: St. Martin's Press, 1980.
- [30] J. W. Richardson and K. A. Scott, "Rap Music and Its Violent Progeny: America's Culture of Violence in Context," *The Journal of Negro Education*, vol. 71, no. 3, pp. 175–192, 2002, doi: 10.2307/3211235.
- [31] R. Shusterman, "The Fine Art of Rap," *New Literary History*, vol. 22, no. 3, pp. 613–632, 1991, doi: 10.2307/469207.
- [32] M. Regev, "Producing Artistic Value: The Case of Rock Music," *The Sociological Quarterly*, vol. 35, no. 1, pp. 85–102, 1994.
- [33] G. Hadjeres, F. Pachet, and F. Nielsen, "DeepBach: a Steerable Model for Bach Chorales Generation," *arXiv:1612.01010 [cs]*, Jun. 2017, Accessed: Dec. 06, 2020. [Online]. Available: <http://arxiv.org/abs/1612.01010>
- [34] D. D. A. Molina, "Adaptive music: Automated music composition and distribution," Universidad de Málaga, Spain, 2016.
- [35] P. Worth and S. Stepney, "Growing Music: Musical Interpretations of L-Systems," in *Applications of Evolutionary Computing*, vol. 3449, F. Rothlauf, J. Branke, S. Cagnoni, D. W. Corne, R. Drechsler, Y. Jin, P. Machado, E. Marchiori, J. Romero, G. D. Smith, and G. Squillero, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 545–550. doi: 10.1007/978-3-540-32003-6_56.
- [36] "The Discrete Emotions Questionnaire: A New Tool for Measuring State Self-Reported Emotions." <https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0159915> (accessed Dec. 06, 2020).
- [37] E. Bigand, "Abstraction of Two Forms of Underlying Structure in a Tonal Melody," *Psychology of Music*, vol. 18, no. 1, pp. 45–59, Apr. 1990, doi: 10.1177/0305735690181004.
- [38] S. L. Chew, L. S. Larkey, S. D. Soli, J. Blount, and J. J. Jenkins, "The abstraction of musical ideas," *Memory & Cognition*, vol. 10, no. 5, pp. 413–423, Sep. 1982, doi: 10.3758/BF03197643.
- [39] M. Duignan, "Computer mediated music production: A study of abstraction and activity," Victoria University of Wellington, 2008. Accessed: Feb. 18, 2021. [Online]. Available: <https://core.ac.uk/download/pdf/41335961.pdf>
- [40] P. Prusinkiewicz and A. Lindenmayer, *The Algorithmic Beauty of Plants*. New York: Springer Verlag, 1990. Accessed: Aug. 20, 2020. [Online]. Available: <http://algorithmicbotany.org/papers/#abop>

- [41] P. Prusinkiewicz, "Graphical applications of L-systems," in *Proceedings of Graphics Interface '86 / Vision Interface '86*, 1986, pp. 247–253.
- [42] E. Costes, C. Smith, M. Renton, Y. Guédon, P. Prusinkiewicz, and C. Godin, "MApplE: Simulation of apple tree development using mixed stochastic and biomechanical models," *Functional Plant Biology*, Mar. 2008, doi: 10.1071/FP08081.
- [43] P. Prusinkiewicz, M. Cieslak, P. Ferraro, and J. Hanan, "Modeling Plant Development with L-Systems," in *Mathematical Modelling in Plant Biology*, R. J. Morris, Ed. Cham: Springer International Publishing, 2018, pp. 139–169. doi: 10.1007/978-3-319-99070-5_8.
- [44] O. Petrenko, O. Terraz, M. Sbert, and D. Ghazanfarpour, "Interactive flower modeling with 3Gmap L-systems," *21st International Conference on Computer Graphics and Vision, GraphiCon'2011 - Conference Proceedings*, Sep. 2011.
- [45] B. F. Lourenço, J. C. L. Ralha, and M. C. P. Brandão, "L-Systems, Scores, and Evolutionary Techniques," in *Proceedings of the SMC 2009 - 6th Sound and Music Computing Conference*, Porto, Portugal, 2009, pp. 113–118.
- [46] B. F. Lourenço, J. C. L. Ralha, and M. C. P. Brandão, "Towards a Genetic L-System Counterpoint Tool," in *12th Brazilian Symposium on Computer Music*, Brazil, 2009, pp. 195–198. Accessed: Aug. 29, 2020. [Online]. Available: <http://compmus.ime.usp.br/sbcm/2009/papers/sbcm-2009-20.pdf>
- [47] M. A. Kaliakatsos-Papakostas, A. Floros, and M. N. Vrahatis, "Intelligent Generation of Rhythmic Sequences Using Finite L-systems," in *2012 Eighth International Conference on Intelligent Information Hiding and Multimedia Signal Processing*, Jul. 2012, pp. 424–427. doi: 10.1109/IIH-MSP.2012.109.
- [48] P. Prusinkiewicz, "Score generation with L-systems," in *Proceedings of the 1986 International Computer Music Conference*, 1986, pp. 455–457.
- [49] A. Rodrigues, E. Costa, A. Cardoso, P. Machado, and T. Cruz, "Evolving L-Systems with Musical Notes," in *Evolutionary and Biologically Inspired Music, Sound, Art and Design*, Cham, 2016, pp. 186–201.
- [50] A. Geambaşu, L. Toron, A. Ravnani, and C. C. Levelt, "Rhythmic Recursion? Human Sensitivity to a Lindenmayer Grammar with Self-similar Structure in a Musical Task," *Music & Science*, vol. 3, p. 2059204320946615, Jan. 2020, doi: 10.1177/2059204320946615.
- [51] J. McCormack, "Grammar Based Music Composition," in *Complex Systems 96*, 1996, pp. 320–336. Accessed: Aug. 29, 2020. [Online]. Available: <https://api.semanticscholar.org/CorpusID:1839457>
- [52] J. von Neumann, *Theory of Self-Reproducing Automata*. University of Illinois Press, 1966.

- [53] C. Bays, "Introduction to Cellular Automata and Conway's Game of Life," in *Game of Life Cellular Automata*, A. Adamatzky, Ed. London: Springer London, 2010, pp. 1–7. doi: 10.1007/978-1-84996-217-9_1.
- [54] E. R. Miranda, "Cellular Automata Music: An Interdisciplinary Project," *null*, vol. 22, no. 1, pp. 3–21, Jan. 1993, doi: 10.1080/09298219308570616.
- [55] A. W. Burks, "Von Neumann's self-reproducing automata," MICHIGAN UNIV ANN ARBOR LOGIC OF COMPUTERS GROUP, 1969.
- [56] E. F. Codd, *Cellular automata*. Academic press, 2014.
- [57] E. R. MIRANDA, "Cellular Automata Music: From Sound Synthesis to Musical Forms," in *Evolutionary Computer Music*, E. R. Miranda and J. A. Biles, Eds. London: Springer London, 2007, pp. 170–193. doi: 10.1007/978-1-84628-600-1_8.
- [58] E. R. Miranda, "Evolving cellular automata music: From sound synthesis to composition."
- [59] D. Burraston and E. Edmonds, "Cellular automata in generative electronic music and sonic art: a historical and technical review," *Digital Creativity*, vol. 16, no. 3, pp. 165–185, 2005.
- [60] M. Solomos, "Cellular automata in Xenakis's music. Theory and Practice," in *International Symposium Iannis Xenakis (Athens, May 2005)*, Greece, 2005, p. 11 p. [Online]. Available: <https://hal.archives-ouvertes.fr/hal-00770141>
- [61] D. Millen, "An interactive cellular automata music application in cocoa."
- [62] J. Kennedy, "Swarm intelligence," in *Handbook of nature-inspired and innovative computing*, Springer, 2006, pp. 187–219.
- [63] A. Abraham, H. Guo, and H. Liu, "Swarm intelligence: foundations, perspectives and applications," in *Swarm intelligent systems*, Springer, 2006, pp. 3–25.
- [64] T. M. Blackwell and P. Bentley, "Improvised music with swarms," in *Proceedings of the 2002 Congress on Evolutionary Computation. CEC'02 (Cat. No. 02TH8600)*, 2002, vol. 2, pp. 1462–1467.
- [65] T. Blackwell, "Swarm music: improvised music with multi-swarms," *Artificial Intelligence and the Simulation of Behaviour, University of Wales*, vol. 10, pp. 142–158, 2003.
- [66] P. Codognet and O. Pasquet, "Swarm intelligence for generative music," in *2009 11th IEEE International Symposium on Multimedia*, 2009, pp. 1–8.
- [67] C. Guéret, N. Monmarché, and M. Slimane, "Ants can play music," in *International Workshop on Ant Colony Optimization and Swarm Intelligence*, 2004, pp. 310–317.

- [68] D. Bisig, M. Neukom, and J. Flury, "Interactive Swarm Orchestra A Generic Programming Environment for Swarm Based Computer Music," 2008.
- [69] X. Zheng *et al.*, "Algorithm composition of Chinese folk music based on swarm intelligence," *International Journal of Computing Science and Mathematics*, vol. 8, no. 5, pp. 437–446, 2017.
- [70] X. Zheng, W. Guo, D. Li, L. Wang, and Y. Wang, "A Hybrid Swarm Composition for Chinese Music," in *International Conference on Swarm Intelligence*, 2017, pp. 258–265.
- [71] M. Allan and C. K. Williams, "Harmonising chorales by probabilistic inference," *Advances in neural information processing systems*, vol. 17, pp. 25–32, 2005.
- [72] A. T. Cemgil, *Bayesian music transcription*. [SI: sn], 2004.
- [73] P. B. Kirlin and D. D. Jensen, "Probabilistic Modeling of Hierarchical Music Analysis.," in *ISMIR*, 2011, pp. 393–398.
- [74] D. Temperley, "A unified probabilistic model for polyphonic music analysis," *Journal of New Music Research*, vol. 38, no. 1, pp. 3–18, 2009.
- [75] L. Rabiner and B. Juang, "An introduction to hidden Markov models," *ieee assp magazine*, vol. 3, no. 1, pp. 4–16, 1986.
- [76] S. R. Eddy, "What is a hidden Markov model?," *Nature biotechnology*, vol. 22, no. 10, pp. 1315–1316, 2004.
- [77] E. Fosler-Lussier, "Markov models and hidden Markov models: A brief tutorial," *International Computer Science Institute*, 1998.
- [78] J.-F. Paiement, D. Eck, and S. Bengio, "A probabilistic model for chord progressions," in *Proceedings of the Sixth International Conference on Music Information Retrieval (ISMIR)*, 2005, no. CONF.
- [79] J.-F. Paiement, "Probabilistic models for music," EPFL, 2008.
- [80] C. Anderson, A. Eigenfeldt, and P. Pasquier, "The generative electronic dance music algorithmic system (GEDMAS)," in *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 2013, vol. 9, no. 1.
- [81] F. Pachet, "The continuator: Musical interaction with style," *Journal of New Music Research*, vol. 32, no. 3, pp. 333–341, 2003.
- [82] G. F. Lawler and V. Limic, *Random walk: a modern introduction*, vol. 123. Cambridge University Press, 2010.
- [83] A. Brown, "Generative music in live performance," in *Generate and Test: Proceedings of the Australasian Computer Music Conference 2005:*, 2005, pp. 23–26.

- [84] C. Roig, L. J. Tardón, I. Barbancho, and A. M. Barbancho, "Automatic melody composition based on a probabilistic model of music style and harmonic rules," *Knowledge-Based Systems*, vol. 71, pp. 419–434, 2014.
- [85] M. H. Hassoun, *Fundamentals of artificial neural networks*. MIT press, 1995.
- [86] W. Liu, Z. Wang, X. Liu, N. Zeng, Y. Liu, and F. E. Alsaadi, "A survey of deep neural network architectures and their applications," *Neurocomputing*, vol. 234, pp. 11–26, 2017.
- [87] D. Eck and J. Schmidhuber, "A first look at music composition using lstm recurrent neural networks," *Istituto Dalle Molle Di Studi Sull Intelligenza Artificiale*, vol. 103, p. 48, 2002.
- [88] M. C. Mozer and T. Soukup, "Connectionist music composition based on melodic and stylistic constraints," in *Advances in Neural Information Processing Systems*, 1991, pp. 789–796.
- [89] F. T. Liang, M. Gotham, M. Johnson, and J. Shotton, "Automatic Stylistic Composition of Bach Chorales with Deep LSTM.," in *ISMIR*, 2017, pp. 449–456.
- [90] B. Sturm, J. F. Santos, and I. Korshunova, "Folk music style modelling by recurrent neural networks with long short term memory units," 2015.
- [91] "abc:standard:v2.1 [abc wiki]." <http://abcnotation.com/wiki/abc:standard:v2.1> (accessed Mar. 07, 2021).
- [92] B. L. Sturm, J. F. Santos, O. Ben-Tal, and I. Korshunova, "Music transcription modelling and composition using deep learning," *arXiv preprint arXiv:1604.08723*, 2016.
- [93] A. Nayebi and M. Vitelli, "Gruv: Algorithmic music generation using recurrent neural networks," *Course CS224D: Deep Learning for Natural Language Processing (Stanford)*, 2015.
- [94] F. Colombo, J. Brea, and W. Gerstner, "Learning to generate music with BachProp," *arXiv preprint arXiv:1812.06669*, 2018.
- [95] L.-C. Yang, S.-Y. Chou, and Y.-H. Yang, "MidiNet: A convolutional generative adversarial network for symbolic-domain music generation," *arXiv preprint arXiv:1703.10847*, 2017.
- [96] K. O'Shea and R. Nash, "An introduction to convolutional neural networks," *arXiv preprint arXiv:1511.08458*, 2015.
- [97] A. Creswell, T. White, V. Dumoulin, K. Arulkumaran, B. Sengupta, and A. A. Bharath, "Generative adversarial networks: An overview," *IEEE Signal Processing Magazine*, vol. 35, no. 1, pp. 53–65, 2018.
- [98] J.-P. Briot, G. Hadjeres, and F.-D. Pachet, "Deep Learning Techniques for Music Generation – A Survey." 2019.

- [99] J.-P. Briot and F. Pachet, "Deep learning for music generation: challenges and directions," *Neural Computing and Applications*, vol. 32, no. 4, pp. 981–993, 2020.
- [100] M. A. Kaliakatsos-Papakostas, A. Floros, and M. N. Vrahatis, "Interactive music composition driven by feature evolution," *SpringerPlus*, vol. 5, no. 1, pp. 1–38, 2016.
- [101] D. Williams *et al.*, "Affective calibration of musical feature sets in an emotionally intelligent music composition system," *ACM Transactions on Applied Perception (TAP)*, vol. 14, no. 3, pp. 1–13, 2017.
- [102] N. C. Maddage, C. Xu, M. S. Kankanhalli, and X. Shao, "Content-based music structure analysis with applications to music semantics understanding," in *Proceedings of the 12th annual ACM international conference on Multimedia*, 2004, pp. 112–119.
- [103] A. Moroni, J. Manzolli, F. V. Zuben, and R. Gudwin, "Vox populi: An interactive evolutionary system for algorithmic music composition," *Leonardo Music Journal*, vol. 10, pp. 49–54, 2000.
- [104] N. Tokui and H. Iba, "Music composition with interactive evolutionary computation," in *Proceedings of the third international conference on generative art*, 2000, vol. 17, no. 2, pp. 215–226.
- [105] L. Bigo, J. Garcia, A. Spicher, and W. E. Mackay, "Papertonnetz: music composition with interactive paper," 2012.
- [106] C. S. Quintana, F. M. Arcas, D. A. Molina, J. D. F. Rodríguez, and F. J. Vico, "Melomics: A case-study of AI in Spain," *AI Magazine*, vol. 34, no. 3, pp. 99–103, 2013.
- [107] A. de la Torre-Luque, R. A. Caparros-Gonzalez, T. Bastard, F. J. Vico, and G. Buela-Casal, "Acute stress recovery through listening to Melomics relaxing music: A randomized controlled trial," *Nordic Journal of Music Therapy*, vol. 26, no. 2, pp. 124–141, 2017.
- [108] F. Pachet and A. R. Addessi, "When children reflect on their own playing style: Experiments with continuator and children," *Computers in Entertainment (CIE)*, vol. 2, no. 1, pp. 14–14, 2004.
- [109] F. Pachet, "Beyond the cybernetic jam fantasy: The continuator," *IEEE Computer Graphics and Applications*, vol. 24, no. 1, pp. 31–35, 2004.
- [110] F. Pachet, "Playing with virtual musicians: The continuator in practice," 2002.
- [111] "Flow Machines – AI assisted music production," *Flow Machines*. <https://www.flow-machines.com/> (accessed Mar. 07, 2021).
- [112] F. Ghedini, F. Pachet, and P. Roy, "Creating music and texts with flow machines," in *Multidisciplinary contributions to the science of creative thinking*, Springer, 2016, pp. 325–343.

- [113] F. Pachet, P. Roy, and B. Carré, "Assisted music creation with Flow Machines: towards new categories of new," in *Handbook of Artificial Intelligence for Music*, Springer, 2021, pp. 485–520.
- [114] F. Pachet, P. Roy, and F. Ghedini, "Creativity through style manipulation: the flow machines project," 2013.
- [115] F. Pachet, A. Papadopoulos, and P. Roy, "Comments on 'Assisted lead sheet composition using FlowComposer'".
- [116] "Dadabots Music." <https://dadabots.com/music.php> (accessed Oct. 17, 2021).
- [117] S. Mehri *et al.*, "SampleRNN: An unconditional end-to-end neural audio generation model," *arXiv preprint arXiv:1612.07837*, 2016.
- [118] Z. Zukowski and C. J. Carr, "Generating Black Metal and Math Rock: Beyond Bach, Beethoven, and Beatles." 2018.
- [119] C. J. Carr and Z. Zukowski, "Generating Albums with SampleRNN to Imitate Metal, Rock, and Punk Bands." 2018.
- [120] C. J. Carr and Z. Zukowski, "Curating Generative Raw Audio Music with D.O.M.E.," in *Joint Proceedings of the ACM IUI 2019 Workshops co-located with the 24th ACM Conference on Intelligent User Interfaces (ACM IUI 2019), Los Angeles, USA, March 20, 2019*, 2019, vol. 2327. [Online]. Available: <http://ceur-ws.org/Vol-2327/IUI19WS-MILC-3.pdf>
- [121] A. Pires and P. Avila, "An approach about the value analysis methodology," *Proceedings of 2100 Projects Association Joint Conferences*, vol. 3, Jan. 2015.
- [122] "AIVA - The AI composing emotional soundtrack music." <https://www.aiva.ai/> (accessed Mar. 07, 2021).
- [123] G. Tanev and A. Božinovski, "Virtual Studio Technology inside Music Production," in *International Conference on ICT Innovations*, 2013, pp. 231–241.
- [124] P. Belliveau, A. Griffin, and S. Somermeyer, *The PDMA toolbox 1 for new product development*. John Wiley & Sons, 2004.
- [125] P. Koen *et al.*, "Providing clarity and a common language to the 'fuzzy front end,'" *Research-Technology Management*, vol. 44, no. 2, pp. 46–55, 2001.
- [126] M. Rose and M. A. Rose, *Writer's block: The cognitive dimension*. SIU Press, 2009.
- [127] C. Haksever, R. Chaganti, and R. G. Cook, "A model of value creation: Strategic view," *Journal of Business Ethics*, vol. 49, no. 3, pp. 295–307, 2004.
- [128] A. Osterwalder, Y. Pigneur, G. Bernarda, and A. Smith, *Value proposition design: How to create products and services customers want*. John Wiley & Sons, 2014.

- [129] C. Larman, *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process*, 2nd ed. USA: Prentice Hall PTR, 2001.
- [130] H. J. BROTHERS, "STRUCTURAL SCALING IN BACH'S CELLO SUITE NO. 3," *Fractals*, vol. 15, no. 01, pp. 89–95, 2007, doi: 10.1142/S0218348X0700337X.
- [131] S. Ornes, "Science and Culture: Hunting fractals in the music of J. S. Bach," *Proc Natl Acad Sci USA*, vol. 111, no. 29, p. 10393, Jul. 2014, doi: 10.1073/pnas.1410330111.
- [132] S. Sukumaran and D. Thiyagarajan, "Generation of fractal music with Mandelbrot set," *Global Journal of Computer Science and Technology*, vol. 9, no. 4, 2009.
- [133] Joint E-mu/Creative Technology Center, "SoundFont 2.1 Application Note." E-mu Systems, Inc., Aug. 12, 1998. Accessed: Oct. 10, 2021. [Online]. Available: <http://freepats.zenvoid.org/sf2/sfapp21.pdf>
- [134] Google, "Flutter." <https://flutter.dev/> (accessed Oct. 10, 2021).
- [135] R. Dreamer, *musicpy*. Accessed: Oct. 10, 2021. [Online]. Available: <https://github.com/Rainbow-Dreamer/musicpy>
- [136] R. Dreamer, *sf2_loader*. Accessed: Oct. 10, 2021. [Online]. Available: https://github.com/Rainbow-Dreamer/sf2_loader
- [137] *iPlug 2*. iPlug 2 Framework, 2021. Accessed: Oct. 10, 2021. [Online]. Available: <https://github.com/iPlug2/iPlug2>
- [138] *juce-framework/JUCE*. JUCE, 2021. Accessed: Oct. 10, 2021. [Online]. Available: <https://github.com/juce-framework/JUCE>
- [139] J. NIELSEN, "Chapter 2 - What Is Usability?," in *Usability Engineering*, J. NIELSEN, Ed. San Diego: Morgan Kaufmann, 1993, pp. 23–48. doi: 10.1016/B978-0-08-052029-2.50005-X.
- [140] N. Chomsky and D. W. Lightfoot, *Syntactic Structures*, 2nd ed. De Gruyter, 2009.
- [141] I. McQuillan, J. Bernard, and P. Prusinkiewicz, "Algorithms for inferring context-sensitive L-systems," in *International Conference on Unconventional Computation and Natural Computation*, 2018, pp. 117–130.
- [142] T. Yokomori, "Stochastic characterizations of EOL languages," *Inf. Control.*, vol. 45, no. 1, pp. 26–33, 1980.
- [143] M. Alfonseca and A. Ortega, "Representation of fractal curves by means of L systems," in *Proceedings of the conference on designing the future*, 1996, pp. 13–21.
- [144] L. Kari, G. Rozenberg, and A. Salomaa, "L systems," in *Handbook of formal languages*, Springer, 1997, pp. 253–328.

- [145] J. Hanan, *Parametric L-systems and their application to the modelling and visualization of plants*. Citeseer, 1992.
- [146] W. W. A. Initiative (WAI), “Web Content Accessibility Guidelines (WCAG) Overview,” *Web Accessibility Initiative (WAI)*. <https://www.w3.org/WAI/standards-guidelines/wcag/> (accessed Oct. 17, 2021).
- [147] “Web Content Accessibility Guidelines (WCAG) 2.1.” <https://www.w3.org/TR/WCAG21/#use-of-color> (accessed Oct. 13, 2021).
- [148] D. Hix and H. R. Hartson, *Developing User Interfaces: Ensuring Usability through Product & Process*. USA: John Wiley & Sons, Inc., 1993.
- [149] “Material Design,” *Material Design*. <https://material.io/> (accessed Oct. 13, 2021).
- [150] “Material Design Components,” *Material Design*. <https://material.io/components?platform=flutter> (accessed Oct. 13, 2021).
- [151] “Widget class - widgets library - Dart API.” <https://api.flutter.dev/flutter/widgets/Widget-class.html> (accessed Oct. 14, 2021).
- [152] “Material Design - Snackbars,” *Material Design*. <https://material.io/components/snackbars> (accessed Oct. 15, 2021).
- [153] J. R. Lewis, “The System Usability Scale: Past, Present, and Future,” *null*, vol. 34, no. 7, pp. 577–590, Jul. 2018, doi: 10.1080/10447318.2018.1455307.
- [154] “Desktop support for Flutter.” <https://flutter.dev/desktop> (accessed Oct. 16, 2021).