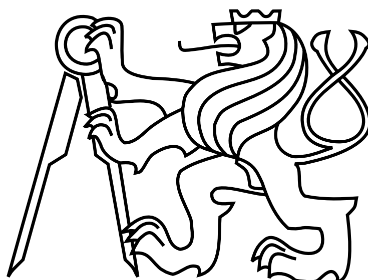
Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Computer Graphics and Interaction



Master's Thesis

Salzella - A Declarative Language for Music Generation

Štěpán Volf

Supervisor: doc. Ing. Karel Richta, CSc.

Study Programme: Open Informatics, Master's Program

Field of Study: Software Engineering

June 9, 2016

Aknowledgements

I would like to express my gratitude to doc. Ing. Karel Richta, CSc. for the time he invested in supervising my thesis. Thanks to my family for supporting me throughout my studies.

Declaration

I declare that I elaborated this thesis on my own and that I mentioned all the information sources and literature that have been used in accordance with the Guideline for adhering to ethical principles in the course of elaborating an academic final thesis.

In Prague on May 26, 2016 ....

Abstract

Salzella is a domain specific declarative language. Its primary focus lies in the field of music generation. The key principle upon which the language is built can be summarized as follows: For any existing piece of music it should be possible to create a Salzella program which will describe it in a way that running this program will output a piece of music similar to the original. Salzella programs are presumed to be generated rather then written by hand. The sole purpose of Salzella is to make creation of music generating tools easier. Bundling Salzella interpreter with music generating software and using it as an engine can provide a significant level of abstraction and thus make the development of such software a lot easier. Tools built on top of Salzella can generate Salzella programs and let the interpreter worry about the actual music generation. The key structural property of Salzella is extensibility. The algorithms responsible for the actual music generation are distributed in a from of plugins. This allows the creators of the aforesaid music generating software to modify Salzella capabilities to their own liking. To prove that Salzella interpreter can indeed be used as internal engine of music generating tools, a prototype of an algorithm for converting musical pieces into Salzella programs was created. As a side effect of creation of this algorithm, three Salzella extensions were created. These extensions are responsible for: generating a melody, generating a simple harmony and generating a percussive complement. An algorithm for harmony analysis was also created as part of this thesis. This algorithm is used by the conversion algorithm. A development environment which allows its users to easily create and execute Salzella programs was also created as part of this thesis.

Abstrakt

Salzella je doménově specifický deklarativní jazyk, jehož využití lze nalézt především v oblasti generování hudby. Klíčový princip, na kterém je jazyk postaven, lze shrnout následovně: Pro jakékoli existující hudební dílo by mělo být možné vytvořit program v jazyku Salzella, který toto dílo popíše tak, že spuštěním tohoto programu dojde k vygenerování hudební díla podobného hudebnímu dílu originálnímu. Hlavním cílem jazyka Salzella je usnadnit vývoj nástrojů pro generování hudby. Nástroje postavené nad platformou Salzella mohou místo hudby generovat programy v jazyku Salzella a samotné generování hudby ponechat na interpretovi tohoto jazyka. Hlavním strukturálním rysem vytvořeného jazyka je rozšiřitelnost.

Algoritmy odpovědné za samotné generování hudby jsou distribuované formou pluginů, což tvůrcům výše zmíněných nástrojů umožňuje rozšířit schopnosti jazyka podle potřeby. Pro ilustraci možnosti tvorby hudbu generujících nástrojů nad platformou Salzella byl vytvořen prototyp algoritmu, který dokáže vytvářet programy v jazyku Salzella na základě existujících hudebních děl. V souvislosti s tvorbou tohoto algoritmu byla vytvořena tři rozšíření jazyka Salzella. Tato rozšíření slouží k: vygenerování melodie, vygenerování jednoduché harmonie a vygenerování perkusivního podkladu. Jakožto součást algoritmu pro převádění hudebních děl do programů Salzella byl vytvořen algoritmus pro analýzu harmonie. Samozřejmou součástí diplomové práce byla rovněž tvorba vývojového prostředí, ve kterém je možné vytvářet a spouštět programy ve vytvořeném jazyku.

Contents

Structure of this document i

1 Introduction 1
- 1.1 Terminology 1
  - 1.1.1 Musical piece 1
  - 1.1.2 Playback 2
  - 1.1.3 Similarity criterion 2
- 1.2 Goal declaration 5
  - 1.2.1 Problem statement 5
  - 1.2.2 Methodology 5
  - 1.2.3 Development environment 6
- 1.3 Related works 7
  - 1.3.1 Open source music libraries 7
  - 1.3.2 MIDI specification 8
- 1.4 Early attempts 8
- 1.5 Name of the language 9

2 Similarity criterion 11
- 2.1 Overall strategy 11
- 2.2 Harmony analysis 12
  - 2.2.1 Problem statement 12
  - 2.2.2 Problem categorization 13
  - 2.2.3 Related works 13
  - 2.2.4 Problem solution 14
  - 2.2.5 Time/space complexity 20
- 2.3 Melody generation 21
  - 2.3.1 Problem statement 21
  - 2.3.2 Problem categorization 22
  - 2.3.3 Related works 22
  - 2.3.4 Problem solution 22
  - 2.3.5 Time/space complexity 26
- 2.4 Percussion generation 26
  - 2.4.1 Problem statement 26
  - 2.4.2 Problem solution 27
  - 2.4.3 Time/space complexity 29

2.5 Specification . . . 29
2.5.1 Parameter . . . 29
2.5.2 Converter . . . 30

3 Extended similarity criterion 33
3.1 Auxiliary definitions . . . 33
3.2 Specification . . . 34
3.2.1 Alphabet . . . 34
3.2.2 Language . . . 35
3.2.3 Parameter . . . 44
3.2.4 Converter . . . 44
3.2.5 Interpreter . . . 46
3.2.6 Extension . . . 47

4 Realization 49
4.1 Interpreter . . . 49
4.1.1 Architecture overview . . . 49
4.1.2 Salzella object model . . . 50
4.1.3 Lightweight MIDI entities . . . 51
4.1.4 Parser . . . 52
4.1.5 Execution control . . . 53
4.1.6 Custom exceptions . . . 53
4.1.7 Utilities . . . 54
4.1.8 Plugin interface . . . 55
4.1.9 Implementation notes . . . 56
4.2 Converter . . . 56
4.3 Plugins . . . 56

5 Development environment 59
5.1 User guide . . . 59
5.1.1 Manual program creation . . . 60
5.1.2 Automated program creation . . . 61
5.1.3 Playback of musical pieces . . . 62
5.1.4 MIDI export/import . . . 63
5.1.5 Performing undo/redo . . . 63
5.2 Realization . . . 64
5.2.1 Architecture . . . 64
5.2.2 Implementation notes . . . 65

6 Conclusion 67
6.1 Evaluation . . . 67
6.1.1 Methodology . . . 67
6.1.2 Algorithms . . . 68
6.1.3 Language . . . 68
6.2 Future work . . . 69
6.2.1 Verse and chorus support . . . 69

6.2.2 Surface matrix convention 69
6.2.3 Manual creation of surface matrices 69

A Lengthy definitions **73**
A.1 Instrument mapping 73
A.2 Percussion mapping 76
A.3 Playback 77
A.4 Integer literals 78
A.5 Literal mappings 78

B Example programs **83**
B.1 Program 1 - Rock 83
B.2 Program 2 - Blues 85
B.3 Program 3 - Jazz 86
B.4 Program 4 - Folk 87
B.5 Program 5 - Classical 89

C UML diagrams **91**
C.1 Salzella object model 91
C.2 Lightweight MIDI entities 92

D Contents of the enclosed CD **93**

Structure of this document

The first three chapters of this thesis deal with formal description of both the problem and its solution. These chapters are addressed primarily to academic audience and are written using strictly formal language. Chapters four and five describe technical realization of the proposed solution. These chapters focus on architectural properties of the developed software and contain important implementation notes. The last chapter evaluates the proposed solution and contemplates on possibilities of future improvements.

Chapter 1: Introduction

In the first chapter, the goal of this thesis will be formally declared. Methodology to achieve the given task will be formulated and related works will be examined. Also, the following three terms will be defined: musical piece, similarity criterion and extended similarity criterion.

Chapter 2: Similarity criterion

The second chapter deals with designing an algorithm which given a musical piece will produce another musical piece similar to the original. This task will be divided into three subtasks: harmony analysis, melody generation and percussion generation. Formally, the task solved in this chapter is creation of an instance of similarity criterion.

Chapter 3: Extended similarity criterion

The third chapter transforms the similarity criterion from the previous chapter into an instance of extended similarity criterion. As a side effect of this transformation a declarative language called Salzella will be created and its formal specification given. The solution proposed in this chapter is extensible in a way which allows it to absorb functionality of arbitrary number of similarity criterions.

Chapter 4: Realization

In the fourth chapter, the implementation of Salzella interpreter will be documented. This includes introducing concepts such as Salzella object model or lightweight MIDI

entities. Contents of this chapter are crucial to anyone who would like to use Salzella interpreter as internal component of music generating tools or create extensions.

Chapter 5: Development environment
In the fifth chapter, Salzella development environment will be described from the software engineering point of view. This chapter provides an overview of architectural properties of the application and contains important implementation notes. A user guide showing how to use the development environment is also part of this chapter.

Chapter 6: Conclusion
The sixth chapter evaluates the proposed solution. Both the advantages and disadvantages of the created language will be discussed. Possibilities of future improvements will be contemplated on.

Appendix A: Lengthy definitions
This appendix contains primarily definitions which require enumerative approach such as functions realizing mapping of integer numbers to types of musical instruments. Instead of unnecessarily prolonging the main body of this document, these definitions were moved to appendix and are referenced from relevant chapters.

Appendix B: Example programs
This appendix contains several example programs written in the created language. Each program addresses a different music genre. The main purpose of these example programs is to illustrate genre independence of the created language.

Appendix C: UML diagrams
This appendix contains UML diagrams which provide a brief overview of structural properties of key components of Salzella platform. UML diagrams for Salzella object model and Salzella lightweight MIDI entities can be found in this appendix.

Appendix D: Contents of the enclosed CD
This appendix contains a graphical overview of contents of the enclosed CD. Both the source codes and executables for all components of Salzella platform can be found on the enclosed CD.

Chapter 1

Introduction

1.1 Terminology

1.1.1 Musical piece

Since this thesis deals with the field of music generation, it will inevitably involve development of music generating algorithms. Regardless of its length or musical quality, the output of a music generating algorithm will be always referred to as a musical piece. In this section a formal definition of what a musical piece is will be provided.

Before the formal definition of musical piece will be provided, two auxiliary definitions will be given. These definitions will introduce concept of events and tracks. Events are formal representations of elementary musical actions such as playing a tone on a piano or hitting a percussive instrument. The concept of tracks allows events to be grouped together based on types of instruments which should be used to realize these events. In most cases, tracks can be thought of as representations of musicians and events as actions performed by these musicians. In addition to a multiset of events and a multiset of tracks, a function which will assign a track to each event will be part of the formal definition of musical piece. Note that this structural concept was borrowed from conventional definitions used in graph theory to assign a pair of vertices to each edge. This structural similarity is emphasized by the notation. Being a usual choice when denoting the incidence function of a graph, ρ will be used to denote the event-to-track assignment function. Note that in order to make the definitions compatible with industry standards, both the lower and upper bounds of integer valued properties were chosen with respect to MIDI specification [1]. There will, however, be some exceptions to this. For example, the range of integer values representing musical instruments will be extended to include value 128 which will be later used to denote percussive instruments.

1.1.1.1 Event

Event is an ordered quadruple of non-negative integers $<start, end, pitch, velocity>$ for which the following constraint holds: $0 \leq start < end < 2^{31} \wedge 0 \leq pitch < 128 \wedge 0 \leq velocity < 128$.

1.1.1.2 Track

Track is an ordered pair of non-negative integers $<instrument, volume>$ for which the following constraint holds: $0 \leq instrument \leq 128 \wedge 0 \leq volume < 128$.

1.1.1.3 Musical piece

Musical piece is an ordered quintuple $<Events, Tracks, \rho, tempo, resolution>$. $Events$ is a multiset of events. $Tracks$ is a multiset of tracks. $\rho : Events \rightarrow Tracks$ is a function which assigns a track to each event. $tempo$ and $resolution$ are positive integer numbers for which the following constraint must hold: $0 < tempo < 2^{31} \wedge 0 < resolution < 2^{31}$.

1.1.2 Playback

Anyone familiar with MIDI specification should find transformation of musical pieces into actual physical sound quite intuitive. Formal description of this transformation will be provided nevertheless. Note, however, that since definitions of functions which assign types of musical instruments and percussive sounds to integer numbers must be handled in enumerative fashion, formal definition of playback is a bit lengthy. And since this definition is not essential, it was moved to appendix A.3.

1.1.3 Similarity criterion

The strategy which will be taken to solve the music generation problem is based on notion of two musical pieces being similar to each other. Providing a mechanism for deciding whether one musical piece is similar to another is therefore necessary. This will be achieved by defining a function which given any musical piece would return a non-empty set of musical pieces. A musical piece m_1 will then be said to be similar to musical piece m_2 if and only if it is contained in a set produced by applying the similarity function on m_2. Note that the binary relation implied by this function will not be required to be symmetric. In other words, a situation where m_1 is similar to m_2 and m_2 is not similar to m_1 will be allowed to occur. Similarly, this relation will not be required to be transitive, ie. if a musical piece m_1 is similar to musical piece m_2 and m_2 is similar to musical piece m_3, m_1 doesn't necessarily have to be similar to m_3. This concept of similarity is obviously awkward. But as discussed later, enforcing these properties could result in great increase in both time and space complexity of the actual implementations of algorithms which would realize this relation. In fact, even the requirement of reflexivity (if attempted to be handled in a non-trivial way) could result in unnecessary increase in complexity and therefore will not be enforced.

When creating an instance of the similarity function, it would be most beneficial to allow its users to influence its behavior by means of parameterizing it with additional data. For example, should some particular implementation of similarity function contain a subroutine which would solve some optimization problem, it could be parametrized with a set of scoring matrices which - depending on their impact on the output - could contain a de facto definition of what makes two musical pieces similar. For sake of simplicity of the formal definition of

similarity function, parameters will be presumed to be provided in a form of a single binary word.

> **1.1.3.1 Similarity criterion**
>
> Let M be a set of all musical pieces. Similarity criterion is an ordered pair $<Parameter, converter>$. $Parameter$ is a subset of all binary words, ie. $Parameter \subseteq \{0,1\}^*$. $converter : M \times Parameter \rightarrow \mathcal{P}(M)$ is a function which for any given musical piece returns a non-empty set of musical pieces.

Every time two musical pieces will be said to be similar, it will be necessary to state what instance of similarity criterion was used when determining the similarity. Similarly, the instance of similarity criterion parameter used to produce the similarity set will be also required to be stated. This is because using a different parameter would most likely result in producing a different similarity set. Also, note that the definition of similarity below is sensitive on order in which the musical pieces are presented. Since the similarity relation is not required to be symmetric, the fact that musical piece $m1$ is similar to $m2$ doesn't necessarily have to mean that $m2$ is similar to $m1$.

> **1.1.3.2 γ-similarity**
>
> Let m_1 and m_2 be two musical pieces. Let γ be an instance of similarity criterion. Let p be a parameter of γ, ie. $p \in \gamma.Parameter$. m_1 is said to be $\gamma(p)$-similar to m_2 if and only if $m_1 \in \gamma.converter(m_2, p)$.

In case the goal of this thesis was to create an algorithm for generating musical pieces similar to musical piece at the input, it could be now said that the goal of thesis is to create and implement an instance of similarity criterion. However, goal of this thesis is to create a language capable of encapsulating essence of musical pieces. Programs written in such language could be either automatically generated based on existing musical pieces or manually written from scratch. This language could be thought of as an 'assembly' language for music generating software creators. Instead of dealing with low-level details of manually creating individual notes, creators of such tools could simply describe the desired contents in higher-level language and let the interpreter of this language worry about the actual music generation. Below is a slightly modified version of similarity criterion which describes this concept formally.

> **1.1.3.3 Extended similarity criterion**
>
> Let M be a set of all musical pieces. Extended similarity criterion is an ordered quintuple $<\Sigma, L, Parameter, converter, interpreter>$. Σ is a non-empty set of alphabet symbols. L is a language over alphabet Σ, ie. $L \subseteq \Sigma^*$. $Parameter$ is a subset of all binary words, ie. $Parameter \subseteq \{0,1\}^*$. $converter : M \times Parameter \rightarrow L$ is a function which for any given musical piece produces a word from L. $interpreter : L \rightarrow \mathcal{P}(M)$ is a function which given a word from L returns a non-empty set of musical pieces.

1.1.3.4 Γ-similarity

Let m_1 and m_2 be two musical pieces. Let Γ be an instance of extended similarity criterion. Let p be a parameter of Γ, ie. $p \in \Gamma.Parameter$. m_1 is said to be $\Gamma(p)$-similar to m_2 if and only if $m_1 \in \Gamma.interpreter(\Gamma.converter(m_2, p))$.

Note that an extended similarity criterion can be created by means of decomposing a similarity criterion. The actual steps taken to achieve this conversion may vary based on particular instances of similarity criterions. The definition below only outlines the general structure of the conversion.

1.1.3.5 Decomposing a similarity criterion

Let γ be a similarity criterion. To decompose γ means to create an extended similarity criterion Γ by reusing $\gamma.Parameter$ and leveraging $\gamma.converter$ to create $\Gamma.\Sigma$, $\Gamma.L$, $\Gamma.converter$ and $\Gamma.interpreter$.

As mentioned earlier, neither γ-similarity nor Γ-similarity is required to be transitive. This is because computational feasibility has to be taken into account. Sets produced by $\gamma.converter$ and $\Gamma.interpreter$ can be huge and their creation computationally very expensive. To overcome this limitation, implementations of these functions will not be required to compute the whole sets. Instead, they will return one random representative of similar musical pieces per execution. Reflecting on this limitation, transitive behavior can be simulated by repeated application of similarity function on such randomly selected musical piece. Note that the following concept of transitive similarity only defines direct, one-step transitivity. Full transitive closure could be also defined. But the definitions below are simpler and more than sufficient for later use.

1.1.3.6 Transitive γ-similarity

Let m_1 and m_2 be two musical pieces. Let γ be an instance of similarity criterion. Let p be a parameter of γ, ie. $p \in \gamma.Parameter$. m_1 is said to be transitively $\gamma(p)$-similar to m_2 if and only if there exists musical piece m_3 such that $m_1 \in \gamma.converter(m_3, p) \wedge m_3 \in \gamma.converter(m_2, p)$.

1.1.3.7 Transitive Γ-similarity

Let m_1 and m_2 be two musical pieces. Let Γ be an instance of extended similarity criterion. Let p be a parameter of Γ, ie. $p \in \Gamma.Parameter$. m_1 is said to be transitively $\Gamma(p)$-similar to m_2 if and only if there exists musical piece m_3 such that $m_1 \in \Gamma.interpreter(\Gamma.converter(m_3, p)) \wedge m_3 \in \Gamma.interpreter(\Gamma.converter(m_2, p))$.

1.2 Goal declaration

1.2.1 Problem statement

The goal of this thesis is to create a declarative language for music generation. The strategy I decided to choose is based on informal observation that whenever a musician comes up with a musical idea, then - assuming this musical idea is conventional enough - it is highly probable that someone else already had the same or at least very similar idea. The idea behind this thesis is to solve the music generation problem by reversing the process of music creation and use already existing musical pieces to create new ones. This will be attempted to achieve by means of creating a language which would allow its users to encapsulate essence of already existing musical pieces in form of programs. If done correctly, execution of such programs should result in generating musical pieces similar to the originals. Note that even though these programs are presumed to be automatically generated based on already existing musical pieces, they can also be written by hand.

To create this language, definition of what makes two musical pieces similar must be created. If only a formal framework for defining what musical similarity is existed. But wait, it does. Creating an instance of extended similarity criterion as defined in 1.1.3.3. will not only provide definition of what makes two musical pieces similar, but as a side effect will also create a language.

1.2.1.1 Problem statement

Create an instance of extended similarity criterion $esc = <\Sigma, L, Parameter, converter, interpreter>$. Provide formal specification for both $esc.\Sigma$ and $esc.L$. Define contents of $esc.Parameter$ and implement $esc.converter$. Implement $esc.interpreter$. Aim to design esc to be versatile in terms of musical genres.

1.2.2 Methodology

Taking advantage of the fact that extended similarity criterions can be created by decomposing similarity criterions, I divided the main task into two subtask. In the first subtask, a similarity criterion will be created. In the second subtask, the similarity criterion from the first subtask will be used as basis for creating an extended similarity criterion. Keeping in mind that there was more than one way to create the similarity criterion from the first subtask, I will aim to make the resulting extended similarity criterion easily adaptable, ie. able to absorb functionality of other possible instances of similarity criterions.

1.2.2.1 Subtask 1

Create an instance of similarity criterion $sc = <Parameter, converter>$. Realize this criterion in a form of prototype, ie. a certain amount of limitations is allowed. For example, this prototype doesn't necessarily have to satisfy the genre independence requirement and can only work with a limited subset of musical genres.

1.2.2.2 Subtask 2

Decompose the similarity criterion from the first subtask and create and implement an extended similarity criterion $esc = <\Sigma, L, Parameter, converter, interpreter>$. Keep in mind that there was more than one way to create the similarity criterion. Aim to make the extended similarity criterion easily adaptable, ie. able to absorb another instances of similarity criterions as well.

In order to satisfy the genre independence requirement, I have decided to aim to design the extended similarity criterion without explicit support of high level structures such as chords, keys and scales and make these possible to define manually within individual programs. As a result, program creators should be fully in charge of what types of chords and scales can be used within their programs and not be forced to choose from a predefined list of options.

1.2.3 Development environment

The secondary goal of this thesis was to create a development environment in which programs written in the created language could be created and executed. A complete list of both functional and non-functional requirements can be found below.

1.2.3.1 Functional requirements

The development environment will allow the user to:

$r1$ - Create Salzella programs

- a - Manually edit code of a Salzella program
- b - Use code completion to paste predefined blocks of code
- c - Request automatic code formatting for better readability

$r2$ - Convert musical pieces into Salzella programs

- a - Select a file containing the musical piece to be converted
- b - Generate a Salzella program by running the conversion algorithm

$r3$ - Validate Salzella programs

$r4$ - Execute Salzella programs

$r5$ - Listen to musical pieces

- a - Select instruments for individual tracks
- b - Adjust volume of individual tracks
- c - Select subset of tracks to be muted
- d - Select subset of tracks to be played back
- e - Adjust tempo of musical piece
- f - Start/stop playback of a musical piece

$r6$ - Load programs from built-in database

$r7$ - Save musical pieces to files

$r8$ - Load musical pieces from files
$r9$ - Perform undo/redo operations over $r1$, $r4$ and $r5a$-$r5e$

When deciding what technology should be used to implement the development environment, I first considered HTML/Javascript. But due to the limited support of sound synthesis and restrictions regarding file system access, I have decided to implement the development environment using the Java programming language instead and thus aim primarily at major desktop platforms.

1.2.3.2 Non-functional requirements

$nr1$ - Use standard MIDI files to store musical pieces
$nr2$ - Implement the development environment using Java SE 8

1.3 Related works

There are dozens of music generating programs and algorithms. Some solutions are robust and aim to generate whole compositions [2], some are designed to perform specific tasks such as generating jazz guitar solos [3]. To some extent, all of them leverage existing principles described in music theory. This often includes outlining the bounds in which a certain level of randomness is introduced. But there are also more exotic approaches like deriving music from cellular automaton patterns [4]. Salzella platform will be designed in a way which will make it possible to integrate existing music generating algorithms in a form of plugins. In other words, Salzella will not compete with other solutions, it will be able to absorb them. No attempts to make such framework seem to have taken place in the past. In order to make the process of absorption of these algorithms as painless as possible, possibilities of adopting some standardized computer representation of musical content were researched. These solutions will be discussed in the rest of this section.

Note Specific subproblems such as harmony analysis or melody generation will be dealt with in this thesis. Related works regarding these topics will be examined in relevant chapters later on. The initial research discussed in this section focused strictly on similar existing solutions and standardized ways of computer representation of musical content.

1.3.1 Open source music libraries

Open source libraries such as [5] or [6] provide their own data types not only for elementary entities such as notes, but also for more complex structures like chords and scales. These libraries could be useful when creating actual music generating algorithms. But there would be little to no benefit from building Salzella on top of any of them. As mentioned above, Salzella will not provide explicit support for high level structures such as chords, scales and keys. This means that the only relevant content of open source music libraries would include several data types and enumerations related to representing a note. Developing these structures from scratch is not a challenging task and the added benefit of being in control over the APIs of these structures is indisputable.

1.3.2 MIDI specification

MIDI specification [1] is primarily a communication protocol designed to deal with real time communication between synthesizers. Several years after its original release in 1983, storage of this communication by means of time stamping individual messages was standardized. Using MIDI specification as basis for structural representation of music seems to be most beneficial. Its popularity and platform independence increases the likelihood of compatibility with musical content representations used by the already existing music generating algorithms. Furthermore, possibility of storing the generated musical contents in MIDI compatible format would allow Salzella users to import the generated snippets of music into external software such as music notation editors or digital audio workstations. Below is an informal description of how conversion of musical pieces into standard MIDI files works.

1.3.2.1 Musical piece → MIDI

Let m be an instance of musical piece. When converting m to a standard MIDI file, start by creating a MIDI sequence and set its tempo and resolution to $m.tempo$ and $m.resolution$, respectively. Next, create a MIDI track for each track in $m.Tracks$. Finally, create a pair of note-on and note-off MIDI messages for each event in $m.Events$ and encapsulate these messages in a MIDI event. Add the event to appropriate track based on $m.\rho$.

1.4 Early attempts

Before diving into specification of the language, let me shortly explain the motivation behind the topic of this thesis. When working on my bachelor thesis, I created a music generating tool called M-Architect [7]. M-Architect was a full featured music editor with additional capabilities such as generating harmonies and bass lines under existing melodies. Shortly after submitting the bachelor thesis, a series of usability tests was performed. In attempt to fix the discovered usability issues, various modifications were made. These included adding support for copy/paste functionality, allowing the user to store/load settings of music generating algorithms or implementing more responsive editor controls. Perhaps the most important improvement, however, was introduction of concept of scenarios. Scenarios allowed the user to predefine complex sequences of actions and store them as macros. The existence of scenarios allowed the application's functionality to be wrapped up and presented in a form of a list of predefined macros and a single generate button.

The aforesaid modifications solved the most critical usability issue: the overall complexity. Unfortunately, where one problem disappeared, another arose. Once advantageous definition of music as triplet <melody, harmony, bass> was suddenly a cause of unnecessary limitations in terms of variety of outputted music. Instead of redesigning the very basis of this particular music generating tool, I decided to create a platform upon which music generating tools could be built. For now, M-Architect is in a dormant state. But in the future, it could be made less restrictive and rebuilt to use Salzella internally.

1.5 Name of the language

The language is named after Mr. Salzella, a character from Terry Pratchett's novel Maskerade [8]. As a reference to the book, an arbitrary number of exclamation marks can be inserted anywhere in a Salzella program without affecting its functionality. Note that this feature is not a part of formal specification of the language. It is an easter egg which only exists in implementation of the language and only applies when invalid programs are passed to the interpreter. If an invalid program is given to the interpreter, it will remove all exclamation marks and attempt to run the program again. Allowing exclamation marks to appear anywhere in Salzella programs at level of formal specification would result in an increase of complexity of the specification. This increase would be surely unjustifiable because this reference to the book is not even that funny.

> **1.5.0.2 Quote from Terry Pratchett's novel Maskerade**
> 'What sort of person,' said Salzella patiently, 'sits down and writes a maniacal laugh? And all those exclamation marks, you notice? Five? A sure sign of someone who wears his underpants on his head. Opera can do that to a man.'

Chapter 2

Similarity criterion

2.1 Overall strategy

In this section, a prototype of an algorithm for generating musical pieces based on existing ones will be created. This algorithm will be formally described as similarity criterion $sc = <Parameter, converter>$ and will represent one of many possible definitions of what makes two musical pieces to be similar. Functionality of this algorithm will be divided into three steps. In the first step, harmony of the original musical piece will be determined. In the second step, a melody will be generated based on contents of one of the melodies of the original musical piece. This melody will be selected by the user. The algorithm presumes that a melody with significance such as main vocal line will be chosen. In the third step, a percussive track will be created. The third track will be created based on a constraint which will be passed to the algorithm in a form of an optional parameter. The reasons for this will be explained in appropriate time.

Note that this algorithm will have many limitations. For example, the outputted musical piece will always consist of at most three tracks. One will contain a melody, second will contain a harmony complement. The optional third track will contain a percussions. The created algorithm aims to capture the essence of the original piece. The presented structural concept of the outputted musical piece seems to do the trick. Arguably, a fourth track containing a bass line should be added. But since this algorithm is meant to be a prototype, generation of the bass line was omitted. Another limitation of the algorithm is that it will only work when dealing with musical genres which presume existence of harmony. But since majority of popular musical styles can be viewed as harmony based, this limitation doesn't seem to be that crucial.

2.1.0.3 Problem decomposition

Let $sc = <Parameter, converter>$ be an instance of similarity criterion which will be described in this section. Functionality of *sc.converter* will consist of the following three steps:

(1) Analyze harmony of the original musical piece and use the result of this analysis to generate a track which will contain a simple chordal complement. The

algorithm used to generate harmony is described in 2.2.

(2) Generate a melody based on contents of one of the tracks from the original musical piece. The track is chosen by the user and harmony detected in the previous step is taken into account. The algorithm used to generate melody is described in 2.3.

(3) Generate a percussive complement which will satisfy the percussion constraint parameter. Both the algorithm used to generate the percussive complement and the percussion constraint are described in 2.4.

2.2 Harmony analysis

2.2.1 Problem statement

Before formally defining the melody harmonization problem, several additional terms will have to be defined. Namely chord, melody and harmony. To make the actual meaning of the following formal definitions easier to grasp, here are few informal notes. Melody is set of events which do not overlap each other in terms of their start/end properties. The same limitation applies on chords which will form a harmony. For the purposes of this thesis two or more different tones sounding at the same time will be considered to be a chord. Also, all tones within a single chord will be required to start and end at the same time and no time gaps will be allowed between chords in chord progressions. Some of the harmony related requirements are quite restrictive and will necessarily result in certain amount of limitation in terms of variety of outputted chord progressions. But again, this algorithm is a part of a prototype of a similarity criterion. The goal at hand is to prove the concept, not to provide a full featured implementation.

2.2.1.1 Melody

Let M be a non-empty set of events. M will be said to be a melody if and only if the following constraint will hold: $\forall a, b \in M : a \neq b \Rightarrow \neg(a.start \leq b.start < a.end)$.

2.2.1.2 Chord

Let C be a set of events containing at least two elements. Let $root$ be a non-negative integer number. An ordered pair $<C, root>$ will be said to be a chord if and only if the following constraint will hold:

(1) $\forall a, b \in C : a \neq b \Rightarrow (a.pitch \bmod 12) \neq (b.pitch \bmod 12)$

(2) $\forall a, b \in C : a.start = b.start \wedge a.end = b.end$

(3) $\exists e \in C : e.pitch = root$

2.2.1.3 Harmony

Let H be a non-empty set of chords. H will be said to be a harmony if and only if the following list of constraints will hold. Note that to make the list of constraints easier to read, the following notation was established: Let c be a chord. $c.start$ will be used to denote *start* value of arbitrary event from $c.C$. Similarly, $c.end$ will be used to denote *end* value of arbitrary event from $c.C$. Since all events in $c.C$ begin and end at the same time, $c.start$ and $c.end$ will effectively denote the time at which c beings and ends, respectively.

(1) $\forall a, b \in H : a \neq b \Rightarrow \neg(a.start \leq b.start < a.end)$

(2) Let $x \in H$ be a chord with minimum value of $x.start$. $\forall a \in H \setminus \{x\} : \exists b \in H : a.start = b.end$

The actual creation of the harmony analysis algorithm will be divided into two steps. In the first step, an algorithm for melody harmonization will be created. This algorithm will be given a melody and produce a harmony suitable to complement the original melody. In the second step, the melody harmonization algorithm will be generalized in a way which will make it possible to pass a whole musical piece at the input instead of a single melody. The main reason behind this decomposition is ease of explanation of the algorithm's functionality. The principle upon which the algorithm is built is much easier to grasp when dealing with single melody line at the input.

2.2.1.4 Problem statement

Create an algorithm which given a melody M will produce a harmony H suitable to complement the melody. Use this algorithm to solve harmony analysis problem, ie. given a musical piece m, create a harmony H_m which would match (or at least closely resemble) the original chordal complement of m.

2.2.2 Problem categorization

Since the task is to find the best of all possible chord progressions, melody harmonization can be thought of as an optimization problem. Music theory offers a number guidelines when it comes to determining the quality of a harmony. However, no universal definition of what optimal harmony is exists. The solution could be also based on enforcing some set of properties which would guarantee a minimal level of musical acceptability. In that case, melody harmonization problem could be viewed as a constraint satisfaction problem. As discussed later, the algorithm created as part of this thesis will solve the task as optimization problem. The absence of existence of the universal definition of optimal solution will be dealt with by delegating the responsibility of providing such definition to the user of the algorithm.

2.2.3 Related works

There are dozens of different solutions of melody harmonization problem. Some of them aim to mimic human composer behavior. For example [9] divides the process of harmony

generation into four parts (segmenting the melody into phrases, assigning harmonic functions to each note, determining a set of possible chords for each note, finding the most agreeable voicing through the chords) and thus aims to emulate human composer cognitive process. There are also solutions which attempt to solve melody harmonization problem by means of genetic algorithms. However, experimental comparison [10] suggests that genetic algorithms can not outperform information-rich strategies.

The most common approach seems to be to view melody harmonization as a constraint satisfaction problem. A good example of this approach is [11]. All possible chords for each tone of the melody are determined. The optimal path is then traced by means of special object representations of constraints which allow the optimal path to be computed efficiently.

2.2.4 Problem solution

What most of the existing solutions seem to have in common is what can be called a functional harmony approach. With respect to the insights of those who formulated the already existing functional harmony based solutions, I have decided to attempt to solve the problem without explicitly referring to functional harmony at all. This means that the algorithm itself will be oblivious of any musical theory guidelines. However, it will be parametrized with a set of scoring matrices capable of encapsulating the essence of functional harmony relations. The quality of results will then be calibratable by means of experimental modifications of the input scoring matrices. The advantage of this approach is that the necessity of expert knowledge required for the algorithm to function properly is delegated to the user of the algorithm. Even though this may seem to be an inconvenience for the user, this approach has its benefits. The users are granted the opportunity to create several different sets of scoring matrices, each one addressing a different musical genre or even compositional style of a particular composer.

This structural concept is inspired by sequence alignment problem. Sequence alignment (one of the basic problems in the field of bioinformatics) is a problem in which two or more sequences of letters are presented. The task is to align these sequences by means of inserting gaps between individual letters. The optimal alignment is then determined with regards to a scoring matrix which is a part of the input and which defines a score for each pair of letters. The optimal solution is one with the highest sum of scores of pairs of letters occupying the same positions within the aligned sequences. While the actual algorithm has nothing to do with melody harmonization, the structural concept is very similar. The algorithm is completely oblivious of the fact that it solves a biological problem. This is because the expert knowledge is abstracted away and moved to the input (creating a scoring matrix) and output (interpreting the results). Also, note that several instances of scoring matrices used in sequence alignment problem were created over the years, each one suitable to be used in different context. The analogy with creating a scoring matrices based on musical genres is hopefully clear and does not require more detailed explanation.

Designing a scoring matrix which could encapsulate the tone/chord/key/progression/... relations is a bit harder than defining relations between pairs of letters. The resulting data structure will be more complex than a single two-dimensional matrix. In the next part of this section, four types of matrices will be presented. First, formal definitions of these matrices

will be given. After the formal definitions will be presented, purpose of each matrix will be explained using an example instance.

2.2.4.1 Key matrix

Let k be a two dimensional matrix. k will be referred to as a key matrix if and only if it satisfies the following list properties:

(1) k has at least one row and exactly twelve columns,

(2) elements of k are integer numbers.

2.2.4.2 Chord matrix

Let c be a two dimensional matrix. c will be referred to as a chord matrix if and only if it satisfies the following list properties:

(1) c has at least one row and exactly twelve columns,

(2) elements of c are either 0 or 1.

2.2.4.3 Segment matrix

Let s be a two dimensional matrix. s will be referred to as a segment matrix if and only if it satisfies the following list properties:

(1) s has exactly one row and at least one column,

(2) elements of s are non-negative integer numbers,

(3) elements of s are ordered in ascending fashion, ie.
$\forall i, j \in \{0, 1, ..., s.numberOfColumns - 1\} : i < j \Rightarrow s[i] < s[j]$.

2.2.4.4 Relation matrix

Let k be a key matrix. Let c be a chord matrix. Let r be a two dimensional matrix. r will be referred to as a kc-relation matrix if and only if it satisfies the following list properties:

(1) number of rows of r is same as number of rows of c,

(2) number of columns of r is same as number of rows of k,

(3) elements of r are sequences of ordered pairs $<root, scoreVector>$ where
- $root$ is an integer number ranging from 0 to 11,
- $scoreVector$ is a sequence of twelve integer numbers.

Before proceeding to formal definition of how the algorithm works, an example of each type of matrix will be given and used to informally explain what individual types of matrices will be used for when the algorithm is ran. Let's begin with the key matrix. An instance of the key matrix will be used to determine a key in which the melody is written. As shown

in the example below, each row of this type of matrix represents one type of a musical key. At the very beginning of the melody harmonization algorithm, twelve passes will be made over the melody for each possible type of key. In each pass, the melody will be transposed accordingly and each tone will be given a score. Key with the highest sum of scores will be selected. Transposition at which the best score was detected will be also stored. Because of its structure, this matrix is capable of detecting subtle variations on standard scales such as usage of harmonic/melodic variations of natural form of major/minor scale. Also, note that it is possible to add support of arbitrary key by simply inserting a new row into the matrix.

2.2.4.5 Example of a key matrix

	0	1	2	3	4	5	6	7	8	9	10	11
major	5	-2	2	-9	8	2	-2	0	2	7	1	7
minor	5	-2	2	8	-9	2	-2	0	4	1	6	2

Next, an instance of the chord matrix will be examined. Each row of this type of matrix represents one type of a chord. Columns containing value 1 mark structure of chords in relation to chromatic scale. Note that when actually implementing this data structure, possibility of creating non-rectangular two dimensional arrays could be leveraged to minimize storage requirements. In this optimized version, rows would be allowed to differ in length and would only contain indexes at which number 1 was present in the unoptimized version. Finally, like with the previous type of matrix, it is possible to add support of arbitrary chord type by simply inserting a new row into the matrix.

2.2.4.6 Example of a chord matrix

	0	1	2	3	4	5	6	7	8	9	10	11
maj	1	0	0	0	1	0	0	1	0	0	0	0
min	1	0	0	1	0	0	0	1	0	0	0	0
aug	1	0	0	1	0	0	1	0	0	0	0	0
dim	1	0	0	0	1	0	0	0	1	0	0	0

Segment matrix is by far the simplest type of matrix there is. This type of matrix carries information about times at which changes in harmony should happen. When the melody harmonization algorithm is ran, each time window implied by the segment matrix will be substituted with a chord. In full featured version of the melody harmonization algorithm, information about when chords in harmony will start and end should be automatically detected. Since this algorithm is a prototype, the rhythmical structure of the harmony must for now be determined manually by the user. However, an algorithm for generating segment matrices could be created and used prior to running the melody harmonization algorithm. This enhancement could be realized without requiring any modifications of the original implementation of the melody harmonization algorithm.

2.2.4.7 Example of a segment matrix

```
         0     1     2     3
time  |  0  | 384 | 576 | 768 |
```

The last type of matrix, the relation matrix, defines relations between keys, chords, and tones of the melody. The meaning of contents of this type of matrix is best explained by an example. The ordered pair `a1` from the example below says that if the melody is in a major key, the tonic chord of this key is given 5 points for any appearance of the first tone of chromatic scale starting at tonic of the key. Appearance of the second tone of this scale would give the tonic chord a penalization of -1. Which makes sense, this tone is not present in any variation of major scale. Appearance of the third tone would yield a score of 2. This tone is in the scale, but is not contained in the tonic chord. That's why it gives a lower score than the first tone which was the root of the tonic chord.

Similarly, the ordered pair `a2` says that if the melody is in a major key, the subdominant chord of this key is given 4 points for any appearance of the first tone of chromatic scale starting at tonic of the key. This also makes sense, this tone is contained within the subdominant chord. Note, however, that this tone plays a role of the perfect fifth within this chord and would receive higher score when considered over the tonic chord where it plays a role of the root.

2.2.4.8 Example of a relation matrix

Auxiliary definitions of ordered pairs $<root, scoreVector>$:

```
a1 = (0,  [5, -1, 1, -4, 4, 1, -4, 4, -2, 1,  0, 2])
a2 = (4,  [4, -1, 1, -4, 4, 0, -3, 0,  5, 1,  0, 2])
a3 = (5,  [4, -2, 1, -4, 1, 5, -1, 1, -4, 7,  2, 0])
a4 = (7,  [2,  1, 6, -4, 0, 2,  1, 5, -1, 1,  1, 4])
a5 = (2,  [1,  0, 5, -4, 4, 4, -1, 1, -2, 7,  1, 1])
a6 = (4,  [0, -4, 1, -1, 2, 0,  0, 1, -3, 1, -1, 2])
a7 = (9,  [4, -4, 1, -1, 4, 1, -1, 1, -1, 5, -1, 2])
a8 = (11, [0, -2, 2, -1, 1, 2, -1, 1, -2, 0, -1, 2])

b1 = (3,  [1,  0, 2, 2, -1, 1, -4, 4, 1, -4,  4, -2])
b2 = (5,  [7, -1, 1, 3, -1, 3, -3, 1, 0,  4,  0, -4])
b3 = (7,  [0, -1, 5, 0, -4, 4, -3, 4, 0,  0, -4,  3])
b4 = (8,  [4,  2, 0, 4, -2, 1, -4, 1, 5, -1,  1, -4])
b5 = (10, [1,  1, 4, 2,  1, 7, -4, 0, 2,  1,  5, -1])
b6 = (0,  [5, -1, 2, 4, -4, 1, -1, 5, 1, -1,  1, -1])
b7 = (5,  [4,  1, 1, 3,  0, 5, -4, 0, 4, -1,  1, -2])
b8 = (7,  [1, -1, 4, 0, -4, 1, -1, 5, 0,  0,  4, -3])
```

```
      major              minor
maj | a1, a2, a3, a4   | b1, b2, b3, b4, b5 |
min | a5, a6, a7       | b6, b7, b8         |
```

```
aug | empty sequence | empty sequence     |
dim | a8             | empty sequence     |
```

In order to make the formal description of melody harmonization algorithm easier to grasp, two auxiliary terms will be defined. First of them will be pitch class. For purposes of harmony analysis, it would make little to no difference if pitches of random subset of input events were transposed by random number of octaves up or down. In most situations, event with pitch $e8$ would have the same impact on harmony properties of the musical piece as $e5$ would have if it substituted the original $e8$. To abstract away differences among pitches which - from harmony point of view - are the same, concept of pitch class will be introduced.

2.2.4.9 Pitch class

Let p be an integer number for which the following constraint holds: $0 \leq p < 128$. Integer number c will be said to be the class of p if and only if $c = p \,\%\, 12$.

The second auxiliary definition deals with concept of transposition. In musical terminology, transposition refers to shifting pitches several semitones higher or lower. For the purposes of this thesis, only upward transpositions limited to one octave shifts will be allowed. Also, note that the formal definition must deal with the fact that a concept of maximum allowed pitch exists. Should the transposition result in creating an invalid pitch, the overflown pitches will be transposed down one octave. The advantage of this solution, as opposed to simply setting the overflown pitch to the maximum allowed value, is that information about class of the overflown pitch will not be lost.

2.2.4.10 Transposition

Let E be a set of events. Let n be an integer number for which the following constraint holds: $0 \leq n < 12$. To transpose E by n semitones means to do the following for each event $e \in E$:

(1) set $e.pitch$ to $e.pitch + n$

(2) if $e.pitch > 127$, set $e.pitch$ to $e.pitch - 12$

Now that all the auxiliary definitions were established, it's time to look at how the melody harmonization algorithm actually works. Due to extensive usage of multi-dimensional matrices, not even usage of pseudo-code would prevent the description of the algorithm from being far to detailed and in terms of semantic meaning human unreadable. I have decided to take more narrative approach in the definitions below. Translating the textual description into pseudo-code should be intuitive. For example, 'considering all combinations of all possible transpositions and musical keys' should be translated as two nested for each loops. The outer loop would iterate over integer numbers ranging from 0 to 11, the inner loop would iterate over rows of a key matrix.

Note Should you find the textual descriptions of algorithms presented in this thesis too vague, refer directly to implementation of the algorithms which can be found on the enclosed

CD. When dealing with algorithms which - at some occasions - require usage of four nested for loops, there seems to be no sensible middle ground in terms of balance between accuracy of description and emphasis of semantic context.

2.2.4.11 Melody harmonization algorithm

Let M be a melody. Let $keyMatrix$ be a key matrix. Let $chordMatrix$ be a chord matrix. Let $segmentMatrix$ be a segment matrix. Let $relationMatrix$ be a relation matrix. To create harmony H suitable to complement melody M do the following:

(1) Initialize H to be an empty set of chords.

(2) Consider all combinations of all twelve possible transpositions of M and all keys defined in $keyMatrix$. Compute a score for each such combination by summing up scores of events of the transposed M. Score for event e and i-th key is equal to value located at i-th row and $(e.pitch \ \% \ 12)$-th column of $keyMatrix$. Find a transposition-key pair which yields the highest score. Store the optimal transposition to $transposition$. Store index of the optimal key to key.

(3) For each segment s implied by $segmentMatrix$, create a set $Q \subseteq M$ which will contain only those events from M which overlap s. An event e overlaps i-th segment if and only if $E \cap S \neq \emptyset$ where E and S are sets built using e and i as defined below.

$$E = \{e.start, e.start + 1, ..., e.end - 1\}$$
$$S = \{segmentMatrix[i], segmentMatrix[i] + 1, ..., segmentMatrix[i + 1] - 1\}.$$

Note that when creating set S for the very last segment, accessing the $(i+1)$-th element would fail because no such column exists. This can be easily fixed by using ∞ instead of $segmentMatrix[i + 1]$ in this particular situation.

(4) Use the $<transposition, key>$ pair obtained in step (2) to identify relevant portion of $relationMatrix$, ie. consider only contents of key-th column of $relationMatrix$. For each set Q created in step (3) iterate over contents of key-th column of $relationMatrix$. Compute a score for each pair $<root, scoreVector>$ by summing up scores of each event from Q. Score for event e is equal to $scoreVector[(e.pitch + transposition) \ \% \ 12]$.

Let p_i denote a $<root, scoreVector>$ pair with maximum score as computed based on contents of i-th set Q. Store index of row of $relationMatrix$ at which p_i was found to $chordType_i$. Store result of $(p_i.root + transposition) \ \% \ 12$ to $chordRoot_i$.

(5) Create a chord c_i for each $<chordType_i, chordRoot_i>$ pair from step (4). Set $c_i.root$ to $chordRoot_i$. Initialize $c_i.C$ to be empty set of events. Let S denote a set of indexes of columns of $chordType_i$-th row of $chordMatrix$ which contain number 1. For each $i \in S$ create an event e and set $e.pitch$ to $chordRoot_i + i$. Set $e.start$ and $e.end$ to match start/end of segment related to c_i. Set $e.velocity$

to 127. Add e to $c_i.C$. Optionally transpose contents of $c_i.C$ several octaves higher. By default, all events of chords in H will be in zeroth or first octave. Add c_i to H.

Note that several interesting modifications could be made to this algorithm. For example, instead of choosing the best scoring key-transposition pair as basis for consequent creation of a chord progression, random key from k best matching keys could be selected. Selecting the key randomly from k best adepts would result in more variety in outputs. Similar trick could be done when choosing the actual chords to fill time windows implied by the segment matrix. Instead of selecting the best chords, random chord could be selected from k best options.

The algorithm for harmonizing melody was created. The task at hand, however, was to create an algorithm which is able to determine harmony of a musical piece. Luckily, the melody harmonization algorithm can be used to analyze harmony of a whole musical piece. The key structural property of set of events which makes it a melody is prohibition of overlapping events. Good news is, that permitting the overlapping of events would not prevent the melody harmonization algorithm from working properly. The melody harmonization algorithm can be therefore turn into harmony analysis algorithm by simply relaxing structural properties which were originally required from the input set of events. The input set of events passed to the harmony analysis algorithm will be created by merging all events from all non-percussive tracks into a single track.

2.2.4.12 Harmony analysis algorithm

Let m be a musical piece. Let $keyMatrix$ be a key matrix. Let $chordMatrix$ be a chord matrix. Let $segmentMatrix$ be a segment matrix. Let $relationMatrix$ be a relation matrix. To determine underlying harmony of m do the following:

(1) Initialize empty set of events S. Add all events from $m.Events$ which are mapped to non-percussive tracks into S. Event e is not mapped to a percussive track if and only if $m.\rho(e).instrument \neq 128$.

(2) Run the melody harmonization algorithm on S, $keyMatrix$, $chordMatrix$, $segmentMatrix$, $relationMatrix$. Present the harmony returned by the melody harmonization algorithm as the result of harmony analysis.

2.2.5 Time/space complexity

Let m denote the number of non-percussive events in the input set. Let h denote the number of chords in harmony. Let k denote the number of possible keys and c the number of possible chord types. Note that h will be always equal to number of segments defined in segment matrix and therefore known prior to running the algorithm. The number of all possible solutions of harmony analysis task is $(12 \cdot c)^h$. Since the number of chords in harmony can be quite big, evaluating all possible solutions is not feasible. The solution space is reduced by analyzing the key in which the musical piece is written first. This operation takes $O(m \cdot k)$. After that, individual segments are processed. The time requiring to process a single segment

depends on number of events which overlap it. The upper bound of processing a single event is $O(m \cdot c)$. Processing all the segments will therefore require $O(h \cdot m \cdot c)$. By simply summing up the complexities of individual steps, the resulting time complexity is $O(m \cdot k + h \cdot m \cdot c)$. Note that the number of key types and chord types is presumed not to vary too much and a reasonable upper bound for these can be found. This upper bound could be used to substitute k and c, turning them into constants. This substitution results in time complexity of $O(m \cdot h)$.

Calculating storage requirements is quite simple and should not require an explanation. The algorithm doesn't require allocation of any extra space which would have asymptotic significance. However, it is important to keep in mind that the input matrices must be stored in the memory throughout the whole execution of the algorithm. Space complexity of the algorithm for harmony analysis is therefore $O(m + h + k \cdot c)$.

> **2.2.5.1 Time/space complexity**
>
> Let m denote the number of non-percussive events in the input set. Let h denote the number of chords in harmony. Let k denote the number of possible keys and c the number of possible chord types.
>
> Time complexity: $O(m \cdot h)$
>
> Space complexity: $O(m + h + k \cdot c)$

2.3 Melody generation

2.3.1 Problem statement

The next algorithm which will be internally used by the similarity criterion deals with problem of melody generation. In particular, the task at hand will be creation of an algorithm which will be able to generate a melody based on contents of another, already existing melody. The formal description below requires the created melody to resemble the original one. Since this algorithm will be part of a similarity criterion, ie. a definition of what makes two musical pieces similar. The notion of resemblance refers to general concept of making the listener to notice the similarity and does not refer to any formally defined concept of similarity. The created algorithm will be part of one of such possible definitions. Note that the following description of melody generation problem builds on top of formal definition of melody provided in 2.2.1.1.

> **2.3.1.1 Problem statement**
>
> Create an algorithm which given a melody M and its underlying harmony H will produce a melody M' which will resemble M and respect the harmony surface implied by H. Allow the user to specify the amount of resemblance by means of adjusting parameters of the algorithm.

2.3.2 Problem categorization

While this problem could be solved as optimization problem, for the purposes of this thesis, the problem will be classified as constraint satisfaction problem. The original melody, its underlying harmony and the additional parameters provided by the user will provide more than enough information to form a set of constraints to be used to outline structural features of the resulting melody.

2.3.3 Related works

The problem of melody generation is one of the fundamental problems in the field of music generation. As such, it has great many solution. Only those solution capable of generating variations on existing melodies were researched. Perhaps the most common approach to generate a variation on an existing melody is to slightly modify its structures by for example adding, removing or randomly shuffling tones and modifying their duration. The more sophisticated solutions employ usage of formal models such as Markov models to introduce the element of randomness. Such models can be built automatically using standard data mining techniques. Good example of this approach is [12].

The similarity criterion created in this chapter will be later decomposed to form an instance of extended similarity criterion. This means that it will be necessary to somehow contain the essence of the original melody in a form of a word over an alphabet. Sure, the exact contents of the original melody could be simply stored in such word and than used as input for the aforesaid algorithms. This would, however, go against the idea of making the created language capable of encapsulating the essence of musical pieces. Stating that the words of the created language would simply contain the original musical pieces would mean that the created language would be merely a file format for storing musical contents. Instead of straightforwardly modifying the original melody, a set of self-contained structural constraints will be created and later made expressible in the created language. Furthermore, the solution proposed in this thesis will leverage the fact, that the algorithm will be given not only the melody, but contextual information in a form of harmony as well.

2.3.4 Problem solution

Receiving a harmony can be useful in determining which tones of the original melody were contained in the underlying chords. The overall consonance of the melody in relation to the harmony can be determined by, for example, simply summing up how many pitches have the same class as at least one pitch contained in the underlying chord. This value could be used as one the constraints to which the generated melody must adhere. Note, however, that it would be also useful to be given information about what scale was used to create the original melody. The harmony analysis algorithm solved this problem using the key matrix, see 2.2.4.11. Since these two algorithms will be merged together, the information obtained by the harmony analysis algorithm can be easily made available to the melody generation algorithm as well. The proposed algorithm will assume it has information about what row of key matrix was selected during analysis of the harmony and what transposition yielded the best score. This information can be used to define set of tones which should be contained in the scale. To express this connection with the harmony analysis algorithm formally, a

parameter containing a set of integer numbers ranging from 0 to 11 will be presumed to be a parameter of the algorithm. This set will be presumed to contain exactly those pitch classes which should be contained in the scale. This set will, however, not be required to be provided by the user. As mentioned earlier, it will be automatically passed to the algorithm from the harmony analysis algorithm.

Note The definition below could be easily modified to allow the generated melodies to use different scales in different time windows. This limitation will be later dealt with by executing the algorithm multiple times over different sections of the original melody.

2.3.4.1 Scale parameter

Let S be a non-empty set of integer numbers such that $S \subseteq \{0, 1, 2, ..., 11\}$. S will be said to be a scale parameter of the melody generation algorithm.

The key idea of the proposed algorithm is to divide the original melody into shorter segments and solve the problem on these segments. Even though this idea is similar to the divide and conquer technique, the problem will not be solved recursively as it usually is the case with algorithms which use this technique. The division will happen only once at the beginning of the algorithm. Individual segments will then be handled by the melody generation algorithm separately, creating multiple shorter melodies which will be later concatenated into one long melody. The actual melody generating algorithm which will be executed on individual segments of the original melody will perform fairly simple statistical analysis and generate a melody which will satisfy the properties of the original segment. Note that by adjusting the length of the segments, the similarity can be increased or decreased. Shorter the segments will be, more similar the melody. Note that the length of segment could vary. But for sake of simplicity, segments of equal length will be presumed. The size of segment will be determined by the user and passed to the algorithm in a form of an integer number as defined below.

2.3.4.2 Segment parameter

Segment parameter is a non-negative integer number.

The statistical analysis will provide a so called structural constraint which will be used to generate the final melody. The structural constraint will provide information about eight properties of the original melody. These include information about duration of an event with the minimal difference between its start and end properties, information about highest/lowest pitch and highest/lowest difference between pitches of consecutive events. They also include information about percentage of tones which were consonant with the underlaying harmony of the original melody. Overall structural properties such as overall direction and rhythmical density of events are part of the structural constraint as well. The following two definitions deal with formally defining what a structural constraint is and what it takes for a melody to satisfy the structural constraint, respectively.

2.3.4.3 Structural constraint

Structural constraint of a melody is an ordered octuple $<grid, min, max, low, high, chord, relax, structure>$.

- $grid$, min, max, low and $high$ are integer numbers for which the following constraint holds: $1 \leq grid < 2^{31} \wedge 0 \leq min \leq max < 128 \wedge 0 \leq low \leq high < 128$.
- $chord$ and $relax$ are non-negative real numbers for which the following constraint holds: $0 \leq chord \leq 1 \wedge 0 \leq relax \leq 1$.
- $structure$ is an order pair $<density, direction>$. $density$ is a sequence of real numbers from the interval $[0, 1]$. $direction$ is a sequence of integer numbers. $direction$ only contains integer numbers from $\{-1, 0, 1\}$. The length of $density$ is equal to length of $direction$.

2.3.4.4 Structural constraint satisfaction

Let M be a melody. M will be said to satisfy structural constraint c if and only if the following list of conditions is satisfied:

(1) $\forall e \in M : e.end - e.start \geq c.grid \wedge e.end - e.start \,\%\, grid = 0$.

(2) For each pair of consecutive events e and e': $c.min \leq |e.pitch - e'.pitch| \leq c.max$.

(3) $\forall e \in M : c.low \leq e.pitch \leq c.high$.

(4) Let $segment$ equal to $c.structure.direction_{length}$. Let e be an event from M which overlaps the i-th segment from equally sized subsegments created by dividing total duration of M by $segment$. If e has a preceding event e', the following constraints must hold. If i-th value from $c.structure.direction$ is equal to -1, then $e.pitch < e'.pitch$. If i-th value from $c.structure.direction$ is equal to 1, then $e.pitch > e'.pitch$. If i-th value from $c.structure.direction$ is equal to 0, then the the relation between $e.pitch$ and $e'.pitch$ can be arbitrary.

Note that neither $structure.density$, $relax$ nor $chord$ properties play a part in determining whether a melody satisfies the structural constraint. These properties are intended to be used as probabilities of certain randomized events. When a melody is being generated, it simply uses real numbers as probabilities of inserting a row at a particular grid slot with $structure.density$ and prolonging duration of an event with $relax$ probability. Not only this version is easier to implement, it also adds more variety to resulting melodies. The melody generation algorithm should be quite intuitive and the formal definition below should not require more detailed explanation. Note, however, that in order to make the description as succinct as possible, some details such as handling of event with pitches which exceed the allowed bounds or handling of special cases in which events have no predecessors were

omitted. To find out how these situations are handled, refer directly to the implementation of the algorithm which can be found on the enclosed CD.

2.3.4.5 Melody generation algorithm

Let M be a melody. Let H be a harmony. Let S be a scale parameter. Let *segmentSize* be a segment parameter. To generate melody M' do the following:

(1) Let e be an event from M. Split M into $\{M_1, M_2, ..., M_n\}$ based on *segmentSize*. Trim *start* and *end* properties of events which overflow the segment size. Create constraint c for each of $\{M_1, M_2, ..., M_n\}$ and generate a melody which satisfies this constraint. Creation of the constraint is described in step (2) of this algorithm. Generating a melody based on the constraint is described in step (3) of this algorithm.

(2) Let M_i denote the currently processed segment. Perform statistical analysis over the melody and create the structural constraint c_i. To calculate $c_i.grid$ find the smallest difference between *start* properties over all pairs of events from M_i. To calculate $c_i.min$ and $c_i.max$ find the minimal and maximal difference between pitches over all consecutive pairs of events from M_i, respectively. To calculate $c_i.low$ and $c_i.high$ find the minimal and maximal pitch of all events from M_i, respectively. To calculate $c_i.chord$, calculate the percentage of how many classes of pitches from all events are equal to at least one pitch class from the underlaying chord. To calculate $c_i.relax$, calculate the percentage of events with duration higher than $c_i.grid$.

    To create *structure*, divide M_i into three segments of equal length and perform statistical analysis of density and direction. Density can be calculated by dividing the number of events by $c_i.grid/3$. If pitch of the first event is lower than pitch of the last event, direction should be equal to 1. If pitch of the first event is higher than pitch of the last event, direction should be equal to -1. If pitch of the first event is the same as pitch of the last event, direction should be equal to 0.

(3) Generate a melody which satisfies constraint c_i from the previous step. Use probabilities defined in $c_i.structure.density$ to decide whether an event should be created when processing grid marks. If the event is created, use $c_i.structure.direction$ to decide whether the pitch of this event should be higher or lower than pitch of its predecessor. Make sure the differences between pitches of consecutive events do not exceed $c_i.min$ and $c_i.max$ bounds and that individual pitches do not exceed $c_i.low$ and $c_i.high$. If the generated pitch is not in chordal consonance with the underlaying harmony, shift it to the closest consonant pitch with the probability of $c_i.chord$. By default, set duration of event to be equal to $c_i.grid$. Prolong the duration of each event with probability of $c_i.relax$. To prolong an event means to set its *end* property to match the start property of the consecutive event if such event exists.

(4) Concatenate the melodies created in step (3) to form the resulting melody.

2.3.5 Time/space complexity

Let m denote the number of events in the original melody. Let *segments* denote the number of segments into which the original melody was split. Splitting the original melody into shorter melody segments will take at most $O(m \cdot segments)$. Since both the statistical analysis of individual melody segments and generation of the resulting melody can be done in linear time in relation to m, splitting the melody is asymptotically the most significant operation. Time complexity of the whole algorithm is therefore $O(m \cdot segments)$.

The current implementation of the algorithm requires $O(m \cdot segments)$ of space because it works in two steps. In the first step, it creates copies of events for each segment. In extreme case of a melody consisting of a single long event, this event will be copied *segments*-times and stored for later processing. However, if the generation of contents of the resulting melody was done upon detection of segments and not postponed to second part of the algorithm, the algorithm would require only $O(m)$ of space.

2.3.5.1 Time/space complexity

Let m denote the number of events in the original melody. Let *segments* denote the number of segments into which the original melody was split.

Time complexity: $O(m \cdot segments)$

Space complexity: $O(m \cdot segments)$ - postponed processing of segments

Space complexity: $O(m)$ - immediate processing of segments

2.4 Percussion generation

2.4.1 Problem statement

The problem of generating a percussive content is seemingly one of the simplest tasks with which the field of music generation deals. Unlike the previous two problems, ie. harmony analysis and melody generation, this task will only have to deal with rhythmical structure of percussive content and not with concepts like scales, keys and chords. The pitch property of events will be used to identify types of percussive hits. For complete overview of mapping of integers values to types of percussive sounds, see A.2. Even though concept of pitches is not relevant when it comes to percussive complement, different types of percussive hits such as snare or kick drum exist. Mutual substitution of these types of percussive sounds could change the musical meaning of the percussive complement dramatically. In order to make the algorithm as simple as possible, percussive complement will be limited to a so called simple drum kit. As shown in the list below, this drum kit will be stripped down to the following four percussive sounds: kick, snare, closed hi-hat and open hi-hat.

2.4.1.1 Simple drum kit

`35` - kick drum

`40` - snare drum

`42` - closed hi-hat

`44` - open hi-hat

2.4.1.2 Rhythm

Let R be a non-empty set of events. R will be said to be a rhythm if and only if the following constraint will hold: $\forall e \in R : e.pitch = 35 \vee e.pitch = 40 \vee e.pitch = 42 \vee e.pitch = 44$.

The problem of percussion generation could be formulated in the same way the melody generation problem was. Given some existing rhythm, the task would be to generate a rhythm which would resemble the original. The analysis of the original rhythm was, however, not realized as part of this thesis. The reasons for this will be obvious later on when it will be shown that for the purposes of the created declarative language, it is more than enough to formulate a rhythm constraint and a mechanism to generate rhythms which will satisfy this constraint. For now, this constraint will be assumed to be passed to the algorithm at the input as a parameter.

2.4.1.3 Problem statement

Create an algorithm which will produce a rhythm R. Allow the user to specify the structure of this rhythm by means of adjusting parameters of the algorithm.

2.4.2 Problem solution

The algorithm used to generate the percussive complement is very simple. The rhythm constraint is designed to make sure that a rhythm contains equally distributed hi-hat hits over the whole the rhythm and snare hits on the explicitly defined places. The remaining properties of the rhythm constraints are used to introduce certain amount of randomness to the resulting rhythm as discussed later on.

2.4.2.1 Rhythm constraint

Rhythm constraint is an ordered sextuple $<loop, hihat, snare, kick, crash, variety>$.

- *loop* and *hihat* are integer numbers for which the following constraint holds: $1 \leq hihat \leq loop < 2^{31}$.
- *snare* is a sequence of integer numbers. The following constraint must hold for each number k in *snare*: $1 \leq k \leq loop$.
- *kick*, *crash* and *variety* are non-negative real numbers for which the following

constraint holds: $0 \leq kick \leq 1 \wedge 0 \leq crash \leq 1 \wedge 0 \leq variety \leq 1$.

2.4.2.2 Rhythm constraint satisfaction

Let R be a rhythm. R will be said to satisfy rhythm constraint c if and only if the following list of conditions is satisfied:

(1) $\forall e \in R : e.pitch = 42 \rightarrow e.end - e.start \geq c.hihat \wedge e.end - e.start \,\%\, hihat = 0$.

(2) $\forall i \in 0..c.loop/c.hihat : \exists e \in R : e.start = i \cdot c.hihat \wedge e.pitch = 42$.

(3) $\forall a \in snare : \exists e \in R : e.pitch = 40 \wedge (e.start = a \vee e.start = a + hihat/2 \vee e.start = a - hihat/2)$.

Note that neither *kick*, *crash* nor *variety* properties play no part in determining whether a rhythm satisfies the rhythm constraint. Just like with the structure constraint which was used in the melody generating algorithm, the properties which are not directly mentioned in the definition of the constraint are intended to be used as probabilities of certain randomized events. When a rhythm is being generated, a kick hit is added at each position where hi-hat hit already exists with probability of *kick*. Similarly, crash hit is added at each position where hi-hat hit already exists with probability of *crash*. After the kicks and crashes are added, *start* and *end* properties of some events are modified according to the rules defined below. The *start* and *end* properties of these events will be modified with probability of *variety*. The rhythm created by the percussion generation algorithm is presumed to be a short beat which is intended to be repeated to create longer percussive complements.

2.4.2.3 Rhythm generation algorithm

Let c be a rhythm constraint. To generate rhythm R do the following:

(1) $\forall i \in 0..c.loop/c.hihat$ create an event with *pitch* property equal to 42. Use i as its *start* and $i + c.hihat$ as its *end* property. Use 127 as *velocity*. Add the event into R.

(2) $\forall a \in snare$ create an event with *pitch* property equal to 40. Use a as its *start* and $a + c.hihat$ as its *end* property. Use 127 as *velocity*. Add the event into R.

(3) Iterate over all hihat events from step (1). For each such event e, generate a kick event with probability of *c.kick*. Use *e.start*, *e.end* and *e.velocity* as its *start*, *end*, and *velocity* properties, respectively. Use 35 as its *pitch* property. Add the event into R.

(4) Do the exact same thing as in step (3) but use *c.crash* as probability of generating an event and 46 as *pitch* of the generated event.

(5) For each event e in R such that $e.pitch = 40$ do the following (with probability of *c.variety*): Subtract or add either *hihat* or $hihat/2$ from *e.start*. Toss a coin

(ie. use probability of 0.5) to decide whether subtraction or addition should take place. Use the same technique to decide whether *hihat* or *hihat*/2 should be subtracted/added.

(6) For each event e in R such that $e.pitch = 42$ do the following (with probability of $c.variety$): Remove e from R with probability of 0.5. For each e which will not be removed during this process, add $hihat/2$ to both $e.start$ and $e.end$. If e will be neither removed nor updated, create event e' with exactly the same properties as e only with both $e.start$ and $e.end$ increased by $hihat/2$ and add e' to R.

2.4.3 Time/space complexity

In each step of the algorithm, all events of R are iterated over. The time spent at each event is constant. Asymptotically speaking, the number of events in R is equal to the number of hi-hat hits created in the first step of the algorithm. This means that time complexity of the percussion generating algorithm is $O(c.loop/c.hihat)$.

The algorithm doesn't require any extra memory usage. The space complexity is equal to the upper bound on number of events which is $O(c.loop/c.hihat)$.

2.4.3.1 Time/space complexity

Let c be a rhythm criterion passed to the percussion generation algorithm.

Time complexity: $O(c.loop/c.hihat)$

Space complexity: $O(c.loop/c.hihat)$

2.5 Specification

The goal of this chapter is to create a similarity criterion, ie. an ordered pair $sc = <Parameter, converter>$. In this section, formal definition of individual components of sc will be given. *Parameter* is a set of binary words, each one representing a valid parameter of the conversion algorithm. The conversion algorithm is formally defined as a function which for any given musical piece returns a non-empty set of musical pieces. This function is named *converter* and it is a key component of sc.

2.5.1 Parameter

Any data which will be passed to *converter* as parameters (such as scoring matrices of the melody harmonization algorithm) must be encodable into a binary word. This requirement is implied by the very paradigm of binary system based computers. Leveraging this implicit requirement, the definition of similarity criterion states that *Parameter* is a set of binary words. Obvious advantage of this definition is its simplicity. Unfortunately, defining a set of all possible parameters in a form of binary words would require defining a whole system for encoding and decoding binary information. This would be extremely tiresome task and the

resulting definition would have no practical use. Instead of defining the actual binary form of parameters, a high level and semantically more sensible approach will be taken and the actual conversion to binary words will be stated to be trivial.

2.5.1.1 Parameter (*Parameter*)

Let k be an instance of key matrix defined in 2.2.4.1. Let c be an instance of chord matrix defined in 2.2.4.2. Let s be an instance of segment matrix defined in 2.2.4.3. Let r be an instance of relation matrix defined in 2.2.4.4. Let rc be a rhythm constraint as defined in 2.4.2.1. Let t be a non-negative integer number. *Parameter* contains exactly those binary words which represent sextuples $<k, c, s, r, rc, t>$. Creation of the actual algorithm used to encode/decode the sextuples into/from binary words is trivial and will not be described in this thesis.

2.5.2 Converter

The functionality of *converter* is built upon the three algorithms described in the first part of this section. The description below is therefore quite straight forward and should not require more detailed explanation. By the definition, the conversion algorithm is supposed to create a set of musical pieces which will be said to be similar to the original musical piece which the algorithm received at the input. But since this set can be huge, its creation may not always be computationally feasible. As mentioned earlier, implementations of this function will not be required to compute the whole sets. Instead, they will return one random representative of similar musical pieces per execution. Note however, that in theory, the whole set of all similar musical pieces could be constructed by executing *converter* repeatedly while systematically ensuring that all possible sequences of random decisions were considered.

Note that in order to generate a percussive track, a rhythm constraint has to be manually passed to the converter. The percussion generation algorithm was created primarily to be absorbed by the extended similarity criterion which will be described in the next chapter. An algorithm for percussive tracks analysis could be created and used to generate the rhythm criterion automatically. Such algorithm, however, was not created as part of this thesis.

2.5.2.1 Converter (*converter*)

Let m be a musical piece. Let $p \in Parameter$. Musical pieces returned by the following randomized algorithm will be said to be similar to m. The algorithm generates one musical piece per execution. To generate a set of all musical pieces similar to m, run the algorithm repeatedly over all possible combinations of outcomes of random events.

(1) Create a musical piece m'. Set values of $m'.tempo$ and $m'.resolution$ to $m.tempo$ and $m.resolution$, respectively. Initialize $m'.Tracks$ to contain three tracks, each one having volume set to 127. Set instrument of the first two tracks to 0, set instrument of the third track to 128.

(2) Decode contents of p and analyze harmony of m by executing the harmony

analysis algorithm described in 2.2. Add all events from the resulting harmony to $m'.Events$ and update $m'.\rho$ to map the events to the first track.

(3) Use the algorithm described in 2.3 to generate a melody similar to the melody selected by the user. Use $p.t$ to determine which track contains this melody. Add all events from the resulting melody to $m'.Events$ and update $m'.\rho$ to map the events to the second track.

(4) Pass rc to the percussion generation algorithm described in 2.4 and generate a percussive loop. Concatenate the loop enough times so that the *end* property of event with the highest *end* property from the rhythm is equal or greater than the *end* property of the event with the highest *end* property from the harmony created in the step (2) of this algorithm. Add all events from the resulting rhythm to $m'.Events$ and update $m'.\rho$ to map the events to the third track.

Chapter 3

Extended similarity criterion

3.1 Auxiliary definitions

So far, the text of this thesis used purely mathematical notation. Since the language created as part of the extended similarity criterion will be a programming language, in this section the purely mathematical description will be gradually abandoned. For example, alphabet Σ will be defined as subset of all characters defined by the ASCII standard [13]. Since the alphabet will contain whitespace characters such as horizontal tabs and line feeds, notation for these symbols must be established. Mostly, whitespace characters will be denoted by escape codes used in C-like programming languages. In order to avoid ambiguity relating to regular space character, an additional escape code `\space` will be defined.

3.1.0.2 Whitespace character escape codes

`\t`	ASCII(9) horizontal tab
`\n`	ASCII(10) line feed
`\f`	ASCII(12) form feed
`\r`	ASCII(13) carriage return
`\space`	ASCII(32) space

There are many ways to formally define a language. Which one is the best, of course, depends on classification of the language in terms of Chomsky hierarchy. As discussed later, the language L which will be a part of the extended similarity criterion is not regular. However, modern derivatives of regular expressions with support of named back referencing such as [14] could be used to specify the language. Due to lack of standardized forms of such regular expressions, this technique will not be used in the documentation of this thesis. But as also discussed later, it will be used in the actual implementation of the validation algorithm which determines whether a word lies in L.

Since the language L is not regular, the next logical step is to move one level up in Chomsky hierarchy and ask: Is the language context-free? Unfortunately, it is not. The reason why the language is not context-free is that a concept similar to named variables is present in the language. This feature allows tracks of a musical piece to be named and use

these names to refer to the tracks later on. Since there is no need to change names of tracks during execution of a program, this concept is limited to 'final' variables only. Meaning that once a named variable references some entity, it can never change to reference another entity. This concept of variables can be easily formulated in a way which is not standard, but very easy to comprehend.

The bottom line is that even though the language is not one hundred percent context-free, it will be treated as such and a derivative of Backus-Naur Form (BNF) [15] will be used to formally define the grammar. As discussed later, the formal specification of the language will be created for documentation purposes only and will not be used to generate the actual parser of the language. Therefore, human readability of the specification is a priority. To make the definitions more readable, `[]` and `{}` notation for optional occurrence and repetition will be borrowed from Extended Backus-Naur form [16]. Note that the following list of symbols is not a complete definition of the notation technique. It serves merely as a reminder for those who are already familiar with both BNF and Extended-BNF. Given the intuitive nature of these notation techniques, anyone who is not familiar with either of them should find this overview sufficient to interpret the meaning of the specification which will be given later on in this chapter.

3.1.0.3 Overview of BNF symbols

`::=`	definition
`|`	alternation
`""`	terminal
`''`	terminal
`{}`	zero or more occurrences (borrowed from Extended-BNF)
`[]`	optional occurrence (borrowed from Extended-BNF)

3.2 Specification

The goal of this thesis is to create and implement an extended similarity criterion, ie. an ordered quintuple $esc = <\Sigma, L, Parameter, converter, interpreter>$. In this section, formal definition of individual components of *esc* will be given. Alphabet sigma will be defined in an enumerative fashion. Language L will defined using a derivative of Backus-Naur form described at the beginning of this chapter. Functionality of *interpreter* will be defined by describing the process of converting word from L to a musical piece. *converter* and *Parameter* were already defined in the previous chapter as part of similarity criterion. Only modifications required to fit these into the extended similarity criterion will be described.

3.2.1 Alphabet

Defining an alphabet is a simple matter of providing an enumerative definition of set of symbols. Alphabet Σ contains 99 symbols all of which can be encoded using ASCII standard. The remaining 29 characters of ASCII encoding which are not contained in Σ are control

characters which have no significance for the purpose of the designed language and were therefore omitted.

3.2.1.1 Alphabet (Σ)

```
Σ = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f, g, h, i,
      j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z, A, B,
      C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U,
      V, W, X, Y, Z, !, ", #, $, %, &, ', (, ), *, +, ,, -, .,
      /, <, =, >, ?, @, [, \, ], ^, _, `, {, |, }, ~, :, ;,
      \t, \n, \f, \r, \space }
```

3.2.2 Language

In this section, the language L will be specified. This will be achieved by defining a set of words over alphabet Σ. Note that these words will also have a semantic meaning which will be described as part of specification of *converter*, ie. a function which given a word from L produces a set of musical piece. From a programmer's point of view, this section deals with syntax only. But the formal definitions will not be thrown at the reader heartlessly and some informal comments about semantic meaning of the declared non-terminals will be made.

As mentioned earlier, a derivative of Backus-Naur form will be used to define the language L. To make the grammar definition easier to understand, it will be presented in sections and each section will be complemented by an informal explanation of its contents. For example, the first part deals with defining a terminal string for each symbol from alphabet Σ. In order to make the further definitions more readable, terminal strings were divided into five groups: digits, letters, special characters, whitespace characters and reserved characters.

3.2.2.1 Grammar (1/12)

```
     <digit> ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" |
                 "7" | "8" | "9"
    <letter> ::= "a" | "b" | "c" | "d" | "e" | "f" | "g" |
                 "h" | "i" | "j" | "k" | "l" | "m" | "n" |
                 "o" | "p" | "q" | "r" | "s" | "t" | "u" |
                 "v" | "w" | "x" | "y" | "z" | "A" | "B" |
                 "C" | "D" | "E" | "F" | "G" | "H" | "I" |
                 "J" | "K" | "L" | "M" | "N" | "O" | "P" |
                 "Q" | "R" | "S" | "T" | "U" | "V" | "W" |
                 "X" | "Y" | "Z"
   <special> ::= "!" | '"' | "#" | "$" | "%" | "&" | "'" |
                 "(" | ")" | "*" | "+" | "," | "-" | "." |
                 "/" | "<" | "=" | ">" | "?" | "@" | "[" |
                 "\" | "]" | "^" | "_" | "`" | "{" | "|" |
                 "}" | "~"
<whitespace> ::= "\t" | "\n" | "\f" | "\r"  | "\space"
  <reserved> ::= ":" | ";"
```

```
<character> ::= <digit> | <letter> | <special> | <whitespace>
```

The next section of the grammar deals with definitions for general purpose literals such as integer numbers, boolean values and strings will be given. Note that the `<integer>` literal could be defined simply as `["-"] <digit> {<digit>}`. However, such definition would imply infinite range of integer numbers. And since integer representations of, for example, musical instruments accept only values from very specific ranges, additional integer literals addressing concrete ranges were created. But because these definitions are a bit lengthy and gloriously ugly, they were moved to A.4.

3.2.2.2 Grammar (2/12)

```
<integer:0-127> ::= see A.4
<integer:0-128> ::= see A.4
<integer:0-INF> ::= see A.4
<integer:1-INF> ::= see A.4
      <boolean> ::= "true" | "false"
       <string> ::= <character> {<character>}
         <word> ::= <letter> {<letter>}
```

Next, definitions of music related literals will be given. Pitch literals represent frequencies of tones. With the total of 128 pitch literals, pitches ranging from C0 to G10 can be addressed. As discussed later, an integer number is associated with each pitch literal. This value can be used as pitch property of events when creating musical pieces. For complete mapping of pitch literals to integer numbers see A.5.

3.2.2.3 Grammar (3/12)

```
  <pitch-class> ::= "c" | "c#" |  "d" | "d#" |  "e" |  "f" |
                   "f#" |  "g" | "g#" |  "a" | "a#" |  "b"
<pitch-literal> ::= <pitch-class> <digit> | "c10" | "c#10" |
                    "d10" | "d#10" | "e10" |"f10" | "f#10" |
                    "g10"
```

Duration literals represent relative durations of tones. Each duration literal consists of two parts: duration base and duration multiplicity. Duration bases represent standard duration symbols used in musical scores. For example, `"4"` refers to quarter note, `"16t"` refers to eighteenth triplet, `"2d"` refers to dotted half note. An integer number is associated with each duration base. For complete mapping of duration bases to integer numbers see A.5. To create a duration literal, duration base must be prefixed with a duration multiplicity. Duration multiplicity is a non-negative integer number followed by symbol 'x'. The actual duration represented by a duration literal will be computed by multiplying the integer representation of the duration base by the duration multiplicity. Values obtained in this manner can be used, for example, as start and end properties of events when creating musical pieces.

3.2.2.4 Grammar (4/12)

```
   <duration-base> ::= "1" |  "1t" |  "1d" |
                       "2" |  "2t" |  "2d" |
                       "4" |  "4t" |  "4d" |
                       "8" |  "8t" |  "8d" |
                      "16" | "16t" | "16d" |
                      "32" | "32t" | "32d" |
                      "64" | "64t" | "64d"
<duration-literal> ::= <integer:0-INF> "x" <duration-base>
```

Note that multiple duration literals can represent the same duration. For example, duration literal `1x2` represents a half note. The duration implied by this literal is exactly the same as duration implied by duration literal `2x4` which represent two quarter notes. Note that duration literal can be also used to identify time at which some event should occur. In this case, duration implied by the duration literal would be interpreted as time elapsed from the beginning of the musical piece. To allow addressing the very beginning of the musical piece, zero length duration literals were introduced. To create a zero length duration literal, simply prefix any duration base with zero multiplicity. Below are a few examples of duration literals.

3.2.2.5 Examples of duration literals

`2x1` - two whole notes
`1x2` - one half note
`4x8` - four eighth notes
`1x64` - one sixty-fourth note
`1x2t` - one half triplet
`3x1d` - three dotted whole notes
`0x4` - zero length duration literal

The fifth section of the grammar definition introduces several types of delimiters which will be used to separate semantically different portions of programs. As should be obvious from the definition below, the created language will be quite benevolent when it comes to whitespace characters. This will allow programmers to format code of their programs to their liking. `":"` and `";"` separators will be used to mark key-value pairs as is so often done in declarative languages. `","` will be used as a line separator in definitions of matrices.

3.2.2.6 Grammar (5/12)

```
   <opt-whitespace> ::= "" | <whitespace> {<whitespace>}
  <space-delimiter> ::= <whitespace> {<whitespace>}
&lt;segment-delimiter&gt; ::= <opt-whitespace> "---" <opt-whitespace>
 <record-delimiter> ::= <opt-whitespace> ";" <opt-whitespace>
    <key-delimiter> ::= <opt-whitespace> ":" <opt-whitespace>
```

```
    <line-delimiter> ::= <opt-whitespace> "," <opt-whitespace>
```

Moving on from the general purpose non-terminals and literals to more complex data structures, three types of matrices will be defined in the next three sections of the grammar. These matrices are used to describe the general structure of the produced musical piece. The overall rhythmic structure to which the generated musical piece will be bound to adhere can be described by creating a signature matrix. Each line of a signature matrix consists of two integer numbers followed by a duration literal. The second integer and the duration literal denote the top and bottom part of a time signature, respectively. The first number denotes how many times should be this time signature applied in the sequence of measures.

3.2.2.7 Grammar (6/12)

```
  <signature-line> ::= <integer:1-INF> <space-delimiter>
                       <integer:1-INF> <space-delimiter>
                       <duration-literal>
<signature-matrix> ::= {<signature-line> <line-delimiter>}
                       <signature-line>
```

The signature matrix shown below says the following: There are 3 measures of 4/4 time signature, then a single measure of 7/8, and then 4 measures of 4/4 time signature.

3.2.2.8 Example of a signature matrix

```
3 4 1x4,
1 7 1x8,
4 4 1x4
```

The next type of matrix defined in the language is called surface matrix. This type of matrix allows the user to specify the tonal surface upon which the musical piece will be built. A surface matrix divides the musical piece into one or more snippets. Each line of this type of matrix consists of a duration literal followed by at least one non-negative integer number. The duration literal defines how long the snippet is. The actual meaning of the integer numbers may vary among individual programs and is in no way predetermined by the specification.

Convention In all example programs created in this thesis, surface matrix is used to determine which scale/chord should be used within individual snippets. To achieve this, the following convention was established: The duration literal in each row is followed by exactly twelve integer numbers. These numbers label individual pitch classes. Pitch classes labeled with 0 are not contained in the scale. Pitch classes labeled with 1 are contained in the scale. Pitch classes labeled with 2 are contained in the scale and are also contained in the chord. Pitch class labeled with 3 is the root of the chord and is also contained in the scale.

3.2.2.9 Grammar (7/12)

```
  <surface-line> ::= <duration-literal> <space-delimiter>
                     <integer:0-INF> {<space-delimiter>
                     <integer:0-INF>}
<surface-matrix> ::= {<surface-line> <line-delimiter>}
                     <surface-line>
```

Surface matrix shown below divides the musical piece into four snippets. All of them are one whole note long and all of them are built on top of the e minor scale. The chord progression defined by this particular surface matrix is: Emi, Dmaj, Cmaj, Bmaj.

3.2.2.10 Example of a surface matrix

```
1x1 1 0 1 0 3 0 1 2 0 1 0 2,
1x1 1 0 3 0 1 0 2 1 0 2 0 1,
1x1 3 0 1 0 2 0 1 2 0 1 0 1,
1x1 1 0 0 2 1 0 2 1 0 1 0 3
```

The last type of matrix defined in the language is called track matrix. This type of matrix defines a list of tracks of which the generated musical piece will be made. Each track is described in a form of a quintuple defining a unique identifier of the track, instrument, volume and mute and solo flags. Instrument and volume are integer numbers, solo and mute are boolean values. Note that the `<word>` non-terminal is used in place where the track identifier should be, meaning that only alphabetical characters can be used in track identifiers.

3.2.2.11 Grammar (8/12)

```
  <track-line> ::= <word> <space-delimiter> <integer:0-128>
                   <space-delimiter> <integer:0-127>
                   <space-delimiter> <boolean>
                   <space-delimiter> <boolean>
<track-matrix> ::= {<track-line> <line-delimiter>}
                   <track-line>
```

Track matrix shown below defines three tracks. All of them are set to be as loud as possible and their solo/mute flags are switched off. The first track uses instrument `73` (flute) and its identifier is `vocal`. Second track uses instrument `0` (piano) and its identifier is `piano`. The last track uses instrument `128` (percussion) and its identifier is `drums`.

3.2.2.12 Example of a track matrix

```
vocal  73 127 false false,
piano   0 127 false false,
drums 128 127 false false
```

Now that the basic data types and data structures are defined, the overall structure of a program can be defined. Each program is dived into segments. There are two types of segments: header segments and filter segments. Each program must contain exactly one header segment and zero or more filter segments. Every program must begin with a header segment. Within this segment, the general structural properties to which the generated musical piece should adhere must be defined. Signature, surface and track matrices must be specified within this segment. Also, a positive integer number must be provided to determine the tempo of the resulting musical piece.

3.2.2.13 Grammar (9/12)

```
<header-segment> ::=
    "tempo" <key-delimiter> <integer:1-INF> <record-delimiter>
  "surface" <key-delimiter> <surface-matrix> <record-delimiter>
"signature" <key-delimiter> <signature-matrix> <record-delimiter>
   "tracks" <key-delimiter> <track-matrix> <record-delimiter>
```

3.2.2.14 Example of a header segment

```
     tempo: 120;
   surface: 1x1 1 0 1 0 3 0 1 2 0 1 0 2,
            2x1 1 0 3 0 1 0 2 1 0 2 0 1,
            1x1 0 2 1 0 2 0 1 1 0 3 0 1,
            1x1 2 0 1 0 2 0 1 1 0 3 0 1,
            2x1 1 0 1 0 3 0 1 2 0 1 0 2,
            3x4 1 0 3 0 1 0 2 1 0 2 0 1,
            1x1 1 0 1 0 3 0 1 2 0 1 0 2,
            1x1 1 0 3 0 1 0 2 1 0 2 0 1,
            1x1 0 2 1 0 2 0 1 1 0 3 0 1,
            1x1 2 0 1 0 2 0 1 1 0 3 0 1,
            7x8 1 0 1 0 3 0 1 2 0 1 0 2,
            7x8 1 0 3 0 1 0 2 1 0 2 0 1;
 signature: 7 4 1x4,
            1 3 1x4,
            4 4 1x4,
            2 7 1x8;
    tracks: vocal  73 127 false false,
            guitar 25 127 false false,
            drums 128 127 false false;
```

Each filter segment has exactly four mandatory properties. The source property decides which plugin should be used when execution of this filter is requested. The input property contains a list of track identifiers determining which tracks will be passed at the input of the filter. Start and duration properties contain duration literals defining a time window at which the filter should be applied. Note that each filter may also define a set of custom properties.

Since parsing of custom properties is handled by filters themselves, there are almost no limitations as to what can be passed to the value part of a custom key-value pair. However, creators of filters are encouraged to use standard pitch and duration literals whenever possible and adhere to the established delimiter convention, ie. use <space-delimiter> as a primary delimiter and <line-delimiter> as a secondary delimiter.

3.2.2.15 Grammar (10/12)

```
    <track-list> ::= <word> {<space-delimiter> <word>}
<filter-segment> ::=
  "source" <key-delimiter> <string> <record-delimiter>
   "input" <key-delimiter> <track-list> <record-delimiter>
   "start" <key-delimiter> <duration-literal> <record-delimiter>
"duration" <key-delimiter> <duration-literal> <record-delimiter>
{<word> <key-delimiter> <string> <record-delimiter>}
```

3.2.2.16 Example of a filter segment

```
    source: cz.stepanvolf.salzella.plugin.GeneralMelody;
     input: guitar;
     start: 4x1;
  duration: 4x1;
      grid: 1x8;
 structure: 0.3 0.5 0.6,
            UP DOWN STEADY STEADY UP;
     chord: 0.7;
     relax: 0.8;
       min: 0;
       max: 2;
       low: e3;
      high: e6;
```

All that remains to do is formally declare the already mentioned fact that a program consists of a single header segment followed by zero or more filter segments. This final step is trivial and should not require a more detailed commentary. Note, however, that every program can both begin and end by optional amount of whitespace characters. It would be highly impractical to classify programs as invalid just because of a few empty lines trailing at their end.

3.2.2.17 Grammar (11/12)

```
<program> ::=
   <opt-whitespace>
   <header-segment> {&lt;segment-delimiter&gt; <filter-segment>}
   <opt-whitespace>
```

Note The example program below uses three types of Salzella filters. All of these filters are external plugins and the underlaying algorithms are not part of the specification of the language. These three algorithms are derived from the algorithms which where created as part of the similarity criterion in Chapter 2. Mapping of these properties to algorithms should be quite intuitive. The three extensions are only simple prototypes. Capabilities of the language are to be extended by creating more sophisticated extensions in the future.

3.2.2.18 Example of a program

```
     tempo: 120;
   surface: 1x1 0 0 1 0 3 0 0 1 0 1 0 2,
            1x1 0 0 3 0 1 0 0 1 0 2 0 1,
            1x1 3 0 1 0 1 0 0 2 0 1 0 1,
            1x1 1 0 1 0 1 0 2 1 0 1 0 3,
            1x1 0 0 1 0 3 0 0 1 0 1 0 2,
            1x1 0 0 3 0 1 0 0 1 0 2 0 1,
            1x1 3 0 1 0 1 0 0 2 0 1 0 1,
            1x1 1 0 1 0 1 0 2 1 0 1 0 3;
 signature: 8 4 1x4;
    tracks: guitar 30 105 false false,
            organ  18  95 false false,
            muted  28  70 false false,
            drums 128 120 false false;
            ---
    source: cz.stepanvolf.salzella.plugin.GeneralMelody;
     input: guitar;
     start: 2x1;
  duration: 2x1;
      grid: 1x8;
 structure: 0.3 0.8,
            UP DOWN UP UP;
     chord: 0.5;
     relax: 0.2;
       min: 0;
       max: 1;
       low: e4;
      high: e5;
            ---
    source: cz.stepanvolf.salzella.plugin.GeneralMelody;
     input: guitar;
     start: 4x1;
  duration: 4x1;
      grid: 1x8;
 structure: 0.3 0.5 0.6,
            UP DOWN STEADY STEADY UP;
     chord: 0.7;
```

```
      relax: 0.8;
        min: 0;
        max: 2;
        low: e3;
       high: e6;
             ---
     source: cz.stepanvolf.salzella.plugin.SimpleDrums;
      input: drums;
      start: 0x1;
   duration: 8x1;
       loop: 1x1;
      hihat: 1x8;
      snare: 1x4 3x4;
       kick: 0.3;
      crash: 0.2;
    variety: 0.1;
             ---
     source: cz.stepanvolf.salzella.plugin.RhythmGuitar;
      input: organ drums;
      start: 0x1;
   duration: 8x1;
       mode: RHYTHM;
       grid: 1x8;
             ---
     source: cz.stepanvolf.salzella.plugin.RhythmGuitar;
      input: muted drums;
      start: 0x1;
   duration: 8x1;
       mode: RHYTHM;
       grid: 1x8;
```

The language L could now be said to contain exactly those words which can be derived from `<program>` non-terminal defined above. However, as mentioned earlier, the language is not one hundred percent context-free because input property of filter segments must contain a list of track names which are declared as a part of the track matrix in the header segment. To create this trivial relation, one additional rule must be declared to ensure that programs referencing non-existent track names will not be contained in L.

3.2.2.19 Grammar (12/12)

For every `<word>` non-terminal w contained in `<track-list>`, a `<word>` non-terminal w' must exist within `<track-matrix>` such that w translates to exactly the same sequence of letters as w'.

3.2.2.20 Language (L)

The set L contains all words over alphabet Σ which can be derived from `<program>` non-terminal defined in 3.2.2.17 and which adhere to rule defined in 3.2.2.19.

3.2.3 Parameter

Since the extended similarity criterion will be created by decomposing the similarity criterion which was created in the second chapter, definition of *Parameter* is very easy. As mentioned in the definition of the similarity criterion decomposition process, defining *Parameter* is a simple matter of reusing *Parameter* of the original similarity criterion. Creation of no special definition is therefore required. *Parameter* is a set of binary words as defined in 2.5.1.1.

3.2.4 Converter

An algorithm which given a musical piece produces a word from L will be defined in this section. To create this algorithm, functionality of the conversion algorithm created in 2.5.2.1 will be decomposed. This will include converting output of the harmony analysis algorithm to surface matrix and creating three types of filter segments which will handle the actual generation of musical content.

3.2.4.1 Converter (*converter*)

Let m be a musical piece. Let $p \in Parameter$. To create a word from L based on contents of m and p, do the following. Begin by creating a `<header-segment>` by following the instructions enclosed in angle brackets in the template below.

`tempo:` <Use tempo of the original musical piece, ie. $m.tempo$> `;`

`surface:` <Run the harmony analysis algorithm from 2.2.4.12 and create a surface matrix based on results of the analysis using the following strategy: Create a string containing a `<duration-literal>` followed by twelve zeros for each chord c in the produced harmony. When concatenating the duration literal and the twelve zeros, use `\space` as delimiter. For each event e in $c.C$, substitute ($e.pitch$ `%` 12)-th zero with 2. Set value of the $c.root$-th number to 3. Let $\langle$key, transposition$\rangle$ be the optimal pair calculated during the key analysis. Do one final iteration over the twelve integer numbers and do the following: If the i-th number from the twelve integer numbers is equal to 0 and $p.keyMatrix$ has a positive integer number at $((i + transposition)$`%`$12)$-th column at key-th row, substitute the 0 with 1. Concatenate all the created strings. Use comma as delimiter.> `;`

`signature:` <Let e be an event from $m.Events$ with maximum value of *end* property. Use integer division when evaluating the following arithmetic expressions. Evaluate $e.end/(3 \cdot m.resolution)$ and store the result to x. Check whether $e.end$ `%` $(3 \cdot m.resolution)$ is equal to zero. If not, increase x by one. The value of x determines how many three-quarter measures should be in the generated musical piece. Note that the conversion algorithm is a prototype which presumes that the overall rhythmical structure is based on three-quarter measures. Full version of this algorithm would of course include automatic detection of rhythmical structure. Even though such algorithm was not created as a part of this thesis, the language takes the possibility of its existence into account. Descriptive capabilities of signature matrices are not limited to three-quarter measure based structures.> `3 1x4;`

```
   tracks: melody    0 127 false false,
           harmony   0 127 false false,
           drums   128 127 false false;
           ---
   source: cz.stepanvolf.salzella.plugin.RhythmGuitar;
    input: harmony;
    start: 0x1;
```

`duration:` <Determine total duration of the musical piece and create a duration literal which expresses this duration.>`;`

```
     mode: STRUM;
     grid: 1x4;
```

For each i from 0 to maximum *end* property in $m.Events$ expressed in whole dotted notes (rounded up), run the melody generation plugin from 2.3.4.5. In the following segment, c will denote structural constraint created in step (2) of the melody generating algorithm.

```
             ---
    source:  cz.stepanvolf.salzella.plugin.GeneralMelody;
     input:  melody;
     start:  0x1;
  duration:  24x16;
```

`grid:` <Express $c.grid$ as a duration literal.>`;`

`structure:` <Express $c.structure_{density}$ as a sequence as comma separated numbers>`,`
<Express $c.structure_{direction}$ as a sequence as comma separated values. Substitute -1 with `DOWN`, 0 with `STEADY` and 1 with `UP`.> `;`

```
        chord:  <Use c.chord.>;
        relax:  <Use c.relax.>;
          min:  <Use c.min.>;
          max:  <Use c.max.>;
          low:  <Express c.low in a form of pitch literal.>;
         high:  <Express c.high in a form of pitch literal.>;
```

3.2.5 Interpreter

Before describing semantic meaning of words from language L, one additional term regarding structure of words from L will be defined. The words from L can thought of as lists of key-value pairs. For example, every word from L must begin with word `tempo` followed by delimiter symbol `":"` followed by an integer number, followed by delimiter symbol `";"`. This structure is often seen in declarative languages. Because the notion of key-value pairs will be used in describing functionality of *interpreter*, formal definition of key value pair will be provided.

Note You may notice that the definition of key-value pair doesn't mention the `";"` delimiter. Since `":"` is reserved, it can not exist in neither key nor value part of a key value pair. As a result, it can be used to unambiguously identify all key-value pairs, ie. existence of `";"` delimiter is not necessary and could be safely removed from the specification of the language. Its presence is, however, a feature which makes Salzella programs more readable. And a bit easier to parse.

3.2.5.1 Key-value pair

Non-terminal *value* will be said to be paired with terminal *key* if and only if *key* is immediately followed by a `<key-delimiter>` and this `<key-delimiter>` is immediately followed by *value*. *value* will be referred to as a value of property *key*.

Interpreter is a function which given a word from L returns a non-empty set of musical pieces. Since computation feasibility must be taken into account, implementation of this function will not return whole set of musical pieces. It will return only one, randomly chosen musical piece from this set. Note the same trick was used when defining the conversion algorithm of the similarity criterion, see 2.5.2.1. Note that the description of this algorithm will not deal with technical aspects. For example, external algorithms will often be loaded and executed during the execution of the program. The technical solution used to achieve this ability is something which will not be dealt with in the specification of the interpreter. It will be left to decide for creators of the its actual implementation. As mentioned earlier, the interpreter will be implemented using the Java programming language. The external algorithms (later referred to as plugins) will be loaded using reflexive capabilities of the language.

3.2.5.2 Interpreter (*interpreter*)

interpreter is a randomized function which given a word from L produces a musical piece. To create a musical piece m from word $w \subset L$, follow the instructions below. Note that this algorithm generates one musical piece per execution. To generate a set of all musical pieces implied by w, run the algorithm repeatedly over all possible combinations of outcomes of random events.

(1) Create a musical piece m and set value of *m.tempo* to value of `"tempo"` from `<header-segment>`. Set and *m.resolution* to 96. Create a track for each `<track-line>` of `<track-matrix>` as defined in `"tracks"` in the `<header-segment>` and set its instrument and volume to `<integer:0-128>` and `<integer:0-127>` from the `<track-line>`, respectively.

(2) For each `<filter-segment>`, execute functionality of plugin defined in its `"source"` property. Before executing the plugin, create a musical piece m' which will be a copy of m but will only contain tracks which were declared in `"input"` property of the `<filter-segment>`. Pass the following data as parameters to the plugin: m', `<surface-matrix>`, `<signature-matrix>`, `"start"`, `"end"` and `"input"` properties as well as all custom properties declared in the `<filter-segment>`. The plugin will produce a musical piece m''. Update m so that the portion implied by `"start"`, `"end"` and `"input"` properties matches contents of m''.

3.2.6 Extension

As mentioned earlier, three Salzella extensions were created as part of this thesis. These three extension are not part of the specification. The overview of their custom properties can be found either in the older version of documentation of this thesis which can found on the enclosed CD, or by invoking the code completion in the development environment.

3.2.6.1 Extensions

`cz.stepanvolf.salzella.plugin.GeneralMelody`
- Generates a simple melody.

`cz.stepanvolf.salzella.plugin.RhythmGuitar`
- Generates a harmony complement.

`cz.stepanvolf.salzella.plugin.SimpleDrums`
- Generates a percussive complement.

Chapter 4

Realization

4.1 Interpreter

4.1.1 Architecture overview

Salzella interpreter consists of seven parts as shown in the overview below. This is how individual components work together: Program execution begins by invoking the parser which creates an instance of Salzella object model. Once the parsing is complete and the Salzella object model created, execution of the program is a fairly straight forward process. The execution begins by creating a lightweight object representation of a MIDI sequence. The overall structure of this sequence is derived from the track matrix. After the sequence is created, filters are applied to it one by one. Given the external nature of these filters, the algorithm responsible for execution is very careful to make defensive copies of the original sequence and checks for unexpected runtime exceptions. In case an error occurs during execution of the program, appropriate exception is thrown. Salzella defines its own exceptions which hold information not only about the kind of error, but also about what segment of program caused it.

4.1.1.1 Salzella interpreter

Salzella interpreter consists of the following seven components:

Salzella object model - A set of classes used to represent Salzella programs. These classes offer convenience methods which allow easy manipulation and derivation of semantic data. More detailed description of Salzella object model will be given in 4.1.2.

Lightweight MIDI entities - A set of classes used to represent musical pieces. Convenience methods for converting the lightweight MIDI sequences into standard MIDI sequences and vice versa are also a part of Salzella platform. More detailed description of lightweight MIDI entities will be given in 4.1.3.

Parser - Implementation of an algorithm which can convert a Salzella program into an instance of Salzella object model. More detailed description of the parser

will be given in 4.1.4.

Execution control - Implementation of an algorithm which can convert an instance of Salzella object model to a musical piece. The musical piece is represented using the lightweight MIDI entities. More detailed description of the interpreter will be given in 4.1.5.

Custom exceptions - Checked exceptions which are thrown in specific situations such as parsing or execution error. More detailed description of custom exceptions will be given in 4.1.6.

Utilities - Useful methods such as validation of Salzella programs or the conversion between Salzella lightweight MIDI sequences and standard MIDI sequences. More detailed description of utilities will be given in 4.1.7.

Plugin interface - An interface to be implemented whenever Salezella plugins are created. More detailed description of the plugin interface will be given in 4.1.8.

4.1.2 Salzella object model

Prior to execution, each program is converted into Salzella object model. The structure of this model closely copies the program structure presented in the previous chapter. Note however, that some of the classes provide convenience methods which can be used when implementing the actual music generating algorithms. For example, the `Segment` class provides a method which can determine whether the given beat is off-beat, on-beat or down-beat In this section, a complete list of all classes will be given. Should you prefer in form a UML digram, see C.1.

4.1.2.1 Salzella object model

Salzella object model consists of the following seven classes, all of which are located in `cz.stepanvolf.salzella.parser` package.

`Program` - Represents a Salzella program. It is an equivalent of the `<program>` non-terminal. It also encapsulates data of the program's `<header-segment>`. Namely it stores information about tempo of the musical piece and references object representations of signature, surface and track matrices.

`Signature` - Represents a surface matrix. It is an equivalent of the `<signature-matrix>` non-terminal. Apart from referencing a list of segments, ie. object representations of `<signature-line>` non-terminals, it offers a convenience method for assigning these segments to integer representations of time.

`Surface` - Represents a surface matrix. It is an equivalent of the `<surface-matrix>` non-terminal. Apart from referencing a list of snippets, ie. object representations of `<surface-line>` non-terminals, it offers a convenience method for assigning these snippets to integer representations of time.

`Container` - Carries information about a musical track. It is an equivalent of the `<track-matrix>` non-terminal. It stores information about identifier, instrument, volume and mute/solo flags of a track.

`Snippet` - Carries arbitrary information related to a particular time window of a musical piece. It is an equivalent of the `<surface-line>` non-terminal. Typically, it stores information about what chord/scale should be used in the particular time window.

`Segment` - Represents a sequence of measures. It is an equivalent of the `<signature-line>` non-terminal. It stores information about the top/bottom part of the time signature and the number of its repetitions.

`Filter` - Represents a filter segment. It is an equivalent of the `<filter-segment>` non-terminal. It stores information about the plugin identifier, the time window at which it should be applied and the list of tracks which should be affected. It also stores all custom properties in a form of a `HashMap`.

4.1.3 Lightweight MIDI entities

Salzella provides its own lightweight implementation of elementary MIDI entities. Note that the lightweight MIDI entities can be useful even in software which is not strictly related to music generation. For example, the development environment created as part of this thesis uses the lightweight implementation of MIDI entities to represent musical contents. Convenience methods for converting Salzella lightweight MIDI entities to standard MIDI format and vice versa can be found in the `Converter` class which is a part of Salzella utilities, see 4.1.7. Should you prefer UML diagram over the textual description below, refer to C.2.

4.1.3.1 Lightweight MIDI entities

Lightweight MIDI entities consists of the following three classes, all of which are located in `cz.stepanvolf.salzella.entity` package.

`Sequence` - Represents a musical piece. It is an equivalent of the standard MIDI sequence. It stores information about tempo of musical piece and references a list of tracks of which this musical piece is made.

`Track` - Represents a track. It is an equivalent of the standard MIDI track. Apart from storing information about instrument and volume, it also stores mute/solo boolean flags.

`Event` - Represents an event. It is an equivalent of the standard MIDI event. The information about the encapsulated note-on and note-off MIDI messages are stored directly in instances of the `Event` class. Therefore, event objects store not only the information about the time (start and end) but pitch and velocity as well.

4.1.4 Parser

As mentioned above, the formal grammar of the language defined in 3.2.2 was not used to generate the parser. The parser was written manually. The advantage of this approach was having the control over the APIs of the parser and the Salzella object model. This is crucial because both the Salzella object model and the parser will be used by creators of Salzella plugins when modifying the contents of relevant tracks and when parsing custom properties. Since the overall structure of a Salzella program can be viewed as a list of key-value pair, parsing a Salzella program is a simple matter of creating a hash map and filling it with key-value pairs.

4.1.4.1 Parser

The functionality of the parser is implemented in a single class. The name of this class is `Parser` and it is located in `cz.stepanvolf.salzella.parser` package. The following static methods are publicly available on this class:

`public static Program parseProgram(String code)`
- Converts the given program into an instance of Salzella object model.

`public static int parseDuration(String code)`
- Returns integer representation of the given duration literal.

`public static int parsePitch(String code)`
- Returns integer representation of the given pitch literal.

The `parseProgram()` method parses structures such as surface or track matrices. The methods which are used to handle these subtasks have private access in the parser because they are used internally.

`private static Surface parseSurface(String code)`
- Converts the given code into an object representation of a surface matrix.

`private static Snippet parseSnippet(String code)`
- Converts the given code into an object representation of a surface matrix line.

`private static Signature parseSignature(String code)`
- Converts the given code into an object representation of a signature matrix.

`private static Segment parseSegment(String code)`
- Converts the given code into an object representation of a signature matrix line.

`private static Container[] parseTracks(String code)`
- Converts the given code into an object representation of a track matrix.

`private static Container parseTrack(String code)`
- Converts the given code into an object representation of a track matrix line.

4.1.5 Execution control

The key responsibility of the execution control is execution of Salzella programs. The algorithm used to execute a Salzella program was described in 3.2.5. The implementation of the execution control receives an instance of Salzella object model at the input. It first creates and instance of the lightweight MIDI sequence and then applies individual filters on the sequence one by one in order in which they were declared.

4.1.5.1 Execution control

The functionality of the execution control is implemented in a single class. The name of this class is `Salzella` and it is located in `cz.stepanvolf.salzella` package. The following static method is publicly available on this class:

`public static Sequence run(String code)`
- Executes the given Salzella program and returns the resulting musical piece.

The execution control can be also invoked indirectly using the `main()` method. This feature makes it possible to execute Salzella interpreter from the command line. The `main()` method is declared in `cz.stepanvolf.salzella.Main` class.

`public static void main(String[] parameters)`
- Uses the first string in `parameters` as path to a file containing a Salzella program. Executes the interpreter and stores the resulting musical piece in a file specified in the second string in `parameters`. The musical piece is stored as a standard MIDI file.

4.1.6 Custom exceptions

Whenever parsing or execution of a Salzella program fails, appropriate exception is thrown. There are three types of exception created in Salzella platform. Each one of them can be thrown when the interpreter is invoked. And since all of these exception are declared as checked, they must be either handled or thrown when the interpreter is executed.

4.1.6.1 Custom exceptions

There are three types of exceptions which can be thrown when the Salzella interpreter is invoked. These exceptions are located in `cz.stepanvolf.salzella.exception` package.

`ParseExcetion` - Thrown whenever parsing of a Salzella program fails.
Example scenario: Some filter segment is missing the `source` property.

`ExecuteExcetion` - Thrown whenever execution of a Salzella program fails.
Example scenario: Runtime exception is thrown when a plugin is executed.

`SequenceExcetion` - Thrown whenever exectution of a Sazella program pro-

duces an invalid musical piece. Example scenario: The track matrix assigns an invalid instrument value to some track.

4.1.7 Utilities

Salzella utilities is a collection of algorithms and constants which can come handy when either creating a Salzella plugins or when creating music generating tools built on top of Salzella platform. The most notable capabilities of this library are validation of Salzella programs and conversion between Salzella MIDI sequences and standard MIDI sequences.

4.1.7.1 Utilities

The following set of classes will be referred to as Salzella utilities. All of these classes are located in `cz.stepanvolf.salzella.util` package.

`Boundary` - Contains constants for lower and upper bounds of integer properties such as pitches and instruments. These constants are used primarily to validate parameters passed to setters and constructors of the lightweight MIDI entities. A few examples of boundary constants: `MIN_PITCH`, `MAX_PITCH`.

`Duration` - Constants in this class are integer representations of duration bases. These constants are used during conversion of duration literals into integer representations of time. A few examples of duration constants: `QUARTER_REGULAR`, `EIGHTH_TRIPLET`.

`Root` - Constants in this class are integer representations of pitch classes. These constants are used during conversion of pitch literals into integer representations of pitch. A few examples of root constants: `C_NATURAL`, `C_SHARP`.

`Volume` - Contains constants for notable volume values. These constants are not used by the Salzella interpreter. They were introduced only to provide some reference values of loudness of events which can be used by creators of plugins. A few examples of volume constants: `F` which stands for forte, `FF` which stands for fortissimo.

`Beat` - Constants in this class represent types of beats. Unlike with the other types of Salzella constants, there is no reason to associate an integer value with a type of a beat. Thats why the types of beats are represented using the built-in support for enumeration pattern. A few example of instances of this enumeration: `ON_BEAT`, `OFF_BEAT`

`Generator` - This class offers convenience methods for generating random values. For example, a static method `randomInteger()` generates random integer numbers from a given range.

`Converter` - Use this class to convert Salzella MIDI sequences to standard MIDI sequences and vice versa. Also, this class offers convenience methods for converting integer value representations of duration and pitch to duration literals

and pitch literals, respectively. These methods are useful when implementing the conversion algorithm.

`Sculptor` - Use this class to perform high level modifications of Salzella sequences such as clearing/trimming all events which overlap a given time window. These methods are used primarily by the execution control of the interpreter. However, they can be useful when creating plugins. For example, the conversion algorithm uses these methods when cutting a melody into segments.

`Validator` - Use this class to validate Salzella programs. Note that for purposes of creating the validation algorithm, the grammar from the formal specification of the language was rewritten using the Java regular expressions. Thanks to usage of back referencing the resulting regular expression can ensure that even the context-sensitive rule which states that the input property of a filter must contain a list of existing track identifiers.

4.1.8 Plugin interface

When dealing with plugins, Salzella interpreter leverages reflexive capabilities of the Java programming language and instead of requiring the source codes of plugins to be available at the compile time, it loads the already compiled plugins on the fly. To make the further manipulation with the plugins easier, each plugin is required to implement the `Plugin` interface which the interpreter uses as a polymorphic wrapper. The `Plugin` interface requires its subclasses to implement a single method named `run()`. Note that the term interface is used here in general sense. `Plugin` is not a Java interface. It is a class with a non-empty constructor which requires its subclasses to ensure that values for instance fields containing information about the plugin such as name of the author are provided when constructing the actual instances.

4.1.8.1 Plugin interface

The `Plugin` class requires its subclasses to implement a single abstract method `run()`. The `Plugin` interface is located in `cz.stepanvolf.salzella.plugin` package. The `run()` method returns a `Sequence` and accepts the following parameters:

`Sequence sequence` - a copy of the original sequence

`Surface surface` - the surface matrix from the header segment

`Signature signature` - the signature matrix from the header segment

`int[] input` - indexes of tracks which should be affected by the filter

`int start` - time at which the time window starts

`int duration` - duration of the time window

`Map<String, String> settings` - custom properties of the filter

4.1.9 Implementation notes

Salzella interpreter was implemented using the Java programming language. Source codes were version controlled using Git. Both the source code and javadoc documentation can be found on the enclosed CD, see Appendix D.

4.2 Converter

The conversion algorithm was implemented as a separate project. Since this algorithm is a prototype, its implementation doesn't always abides the good practices a proper software engineer should adhere to. For example, instead of being loaded from an external file, the scoring matrices used by the harmony analysis algorithm are hard coded in the actual implementation. The conversion algorithm was implemented using the Java programming language and version controlled using Git. Both the source code and javadoc documentation can be found on the enclosed CD. For complete overview of contents of the enclosed CD see Appendix D.

4.2.0.1 Converter

The functionality of the conversion algorithm is spread across two classes, both of which are located in `cz.stepanvolf.binky` package.

`Binky` - Implementation of the actual conversion algorithm. The only public method in this class is named `run()`. It accepts a MIDI file at the input and produces a string representation of a Salzella program. `ConvertException` is thrown in case the conversion fails.

`Matrix` - Static fields of this class contain scoring matrices used by the harmony analysis algorithm. In full version of the algorithm, these matrices should be loaded from a file instead of being hard coded in this class.

4.3 Plugins

Three plugins were created as part of this thesis. These plugins were implemented using the Java programming language and version controlled using Git. Both the source code and javadoc documentation can be found on the enclosed CD. For complete overview of contents of the enclosed CD see Appendix D.

4.3.0.2 Plugins

Three plugins were created as part of this thesis. All of these classes are located in `cz.stepanvolf.salzella.plugin` package.

`GeneralMelody` - Implementation of the melody generating filter.

`RhythmGuitar` - Implementation of the rhythm guitar generating filter.
`SimpleDrums` - Implementation of the percussion generating filter.

Chapter 5

Development environment

5.1 User guide

A development environment for creating Salzella programs was created as part of this thesis. The user interface of Salzella development environment consists of a source code editor, playback controls and three Salzella control buttons. These buttons allow the user to run a Salzella program, load an existing Salzella program from a built-in database and generate a program based on contents of a MIDI file. Usage of the development environment is quite intuitive. The key use case scenarios will be described in greater detail nonetheless. It will be shown that combined together, the individual use case scenarios cover all functional requirements defined in 1.2.3. Figure 5.1 shows and overview of the user interface.

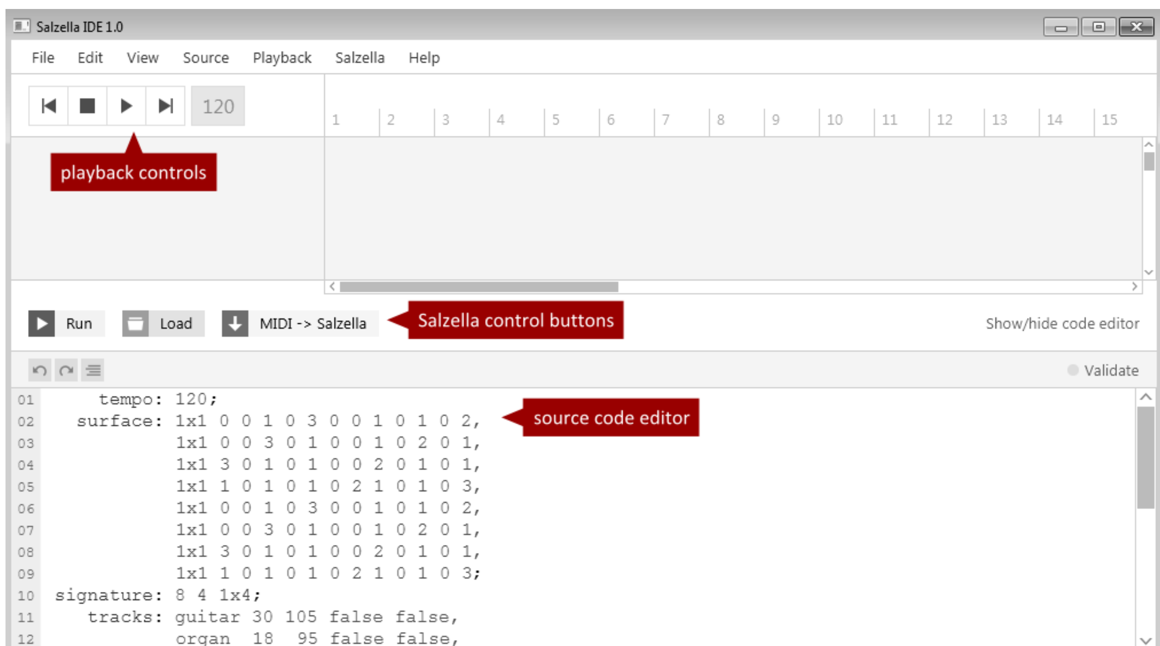


Figure 5.1: Salzella development environment

5.1.1 Manual program creation

Figure 5.2 uses labels *c*1-*c*7 to mark controls relevant to manual creation of Salzella programs. To manually create a Salzella program, use the keyboard to edit contents of text area *c*1. When editing the source code of a Salzella program, use buttons *c*2/*c*3 to undo/redo changes, click button *c*4 to automatically format the code for better readability, click *c*5 to validate the program. Use *c*6 to request code completion. When done editing the source code, the program can be executed by clicking button *c*7. The execution of the program will result in creation of a musical piece.

5.1.1.1 Functional requirements coverage (1/5)

Controls *c*1-*c*7 are related to the following functional requirements:

*r*1*a* - Manually edit code of a Salzella program

*r*1*b* - Use code completion to paste predefined blocks of code

*r*1*c* - Request automatic code formatting for better readability

*r*3 - Validate Salzella programs

*r*4 - Execute Salzella programs

*r*9 - Perform undo/redo operations over *r*1, *r*4 and *r*5*a*-*r*5*e*

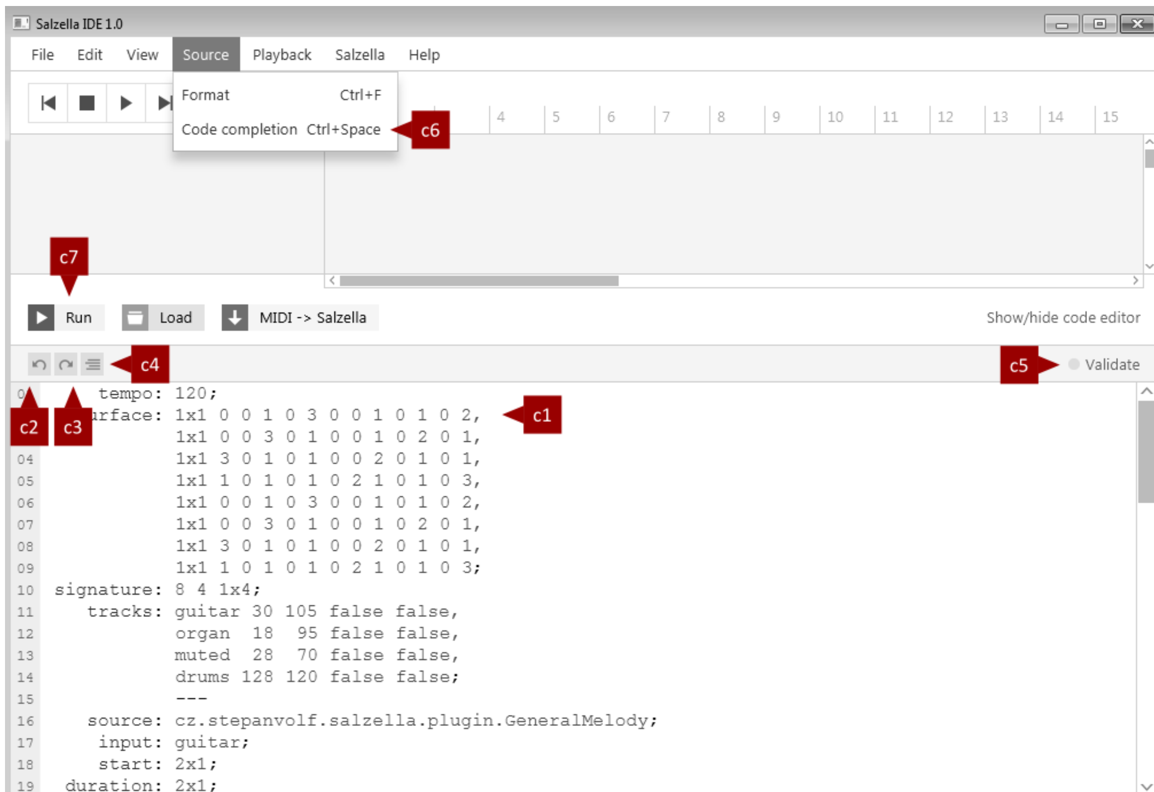


Figure 5.2: Manual program creation

5.1.2 Automated program creation

Figure 5.3 uses labels *c*8-*c*9 to mark controls relevant to automated creation of Salzella programs. To run the conversion algorithm click on button *c*8 and use the dialog window to locate a MIDI file. After selecting a file, the algorithm will be executed and the resulting Salzella program automatically loaded into the code editor. To load a Salzella program from built-in database, click on button *c*9 and select a program from the presented list.

5.1.2.1 Functional requirements coverage (2/5)

Controls *c*8-*c*9 are related to the following functional requirements:

*r*2*a* - Select a file containing the musical piece to be converted

*r*2*b* - Generate a Salzella program by running the conversion algorithm

*r*6 - Load programs from built-in database

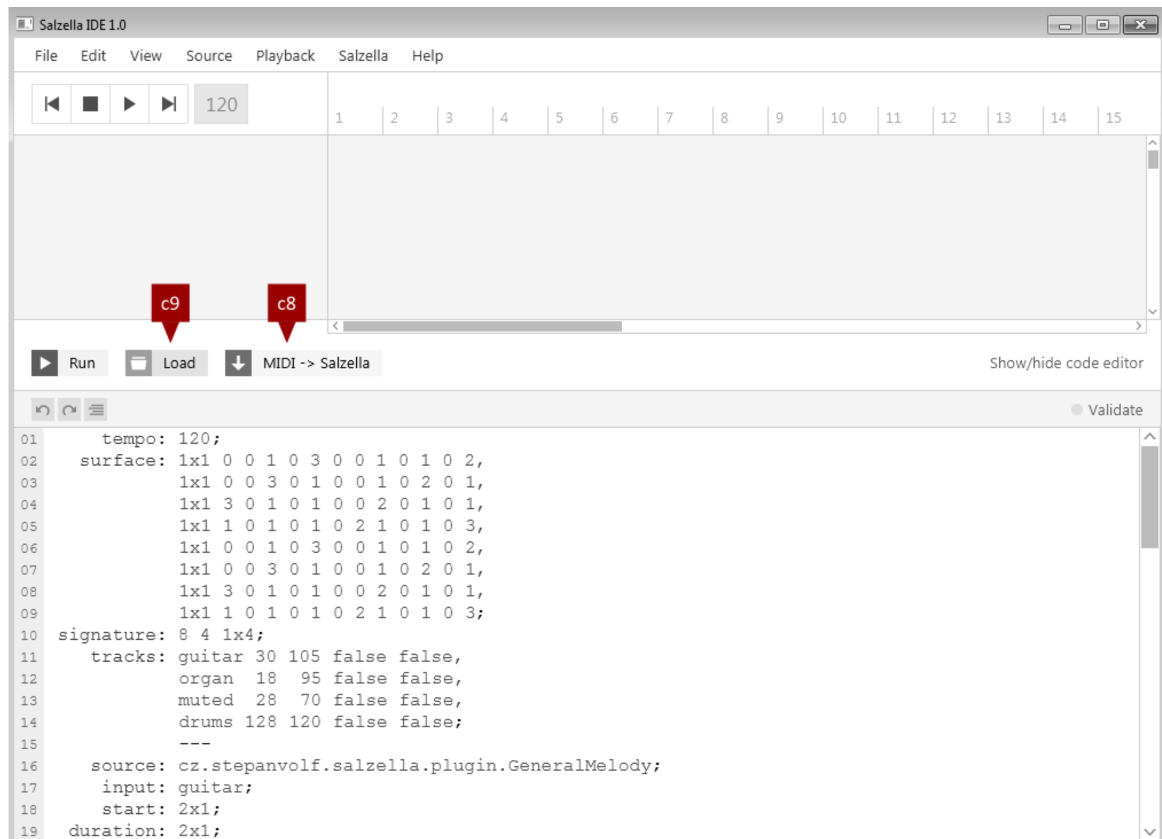


Figure 5.3: Automated program creation

5.1.3 Playback of musical pieces

Figure 5.4 uses labels $c10$-$c18$ to mark controls relevant to playback of musical pieces. Whenever a musical piece is generated, it can be played back by clicking button $c10$. The playback can be stopped by clicking button $c11$. To change tempo of the musical piece, use the keyboard to edit contents of text area $c12$. To change instrument of a track, use button $c13$. To adjust volume of a track, use slider $c14$. To change mute/solo flags, use buttons $c15/c16$. To expand vertical space allocated for musical piece visualization, hide the code editor by clicking button $c18$.

5.1.3.1 Functional requirements coverage (3/5)

Controls $c10$-$c18$ are related to the following functional requirements:

$r5a$ - Select instruments for individual tracks

$r5b$ - Adjust volume of individual tracks

$r5c$ - Select subset of tracks to be muted

$r5d$ - Select subset of tracks to be played back

$r5e$ - Adjust tempo of musical piece

$r5f$ - Start/stop playback of a musical piece

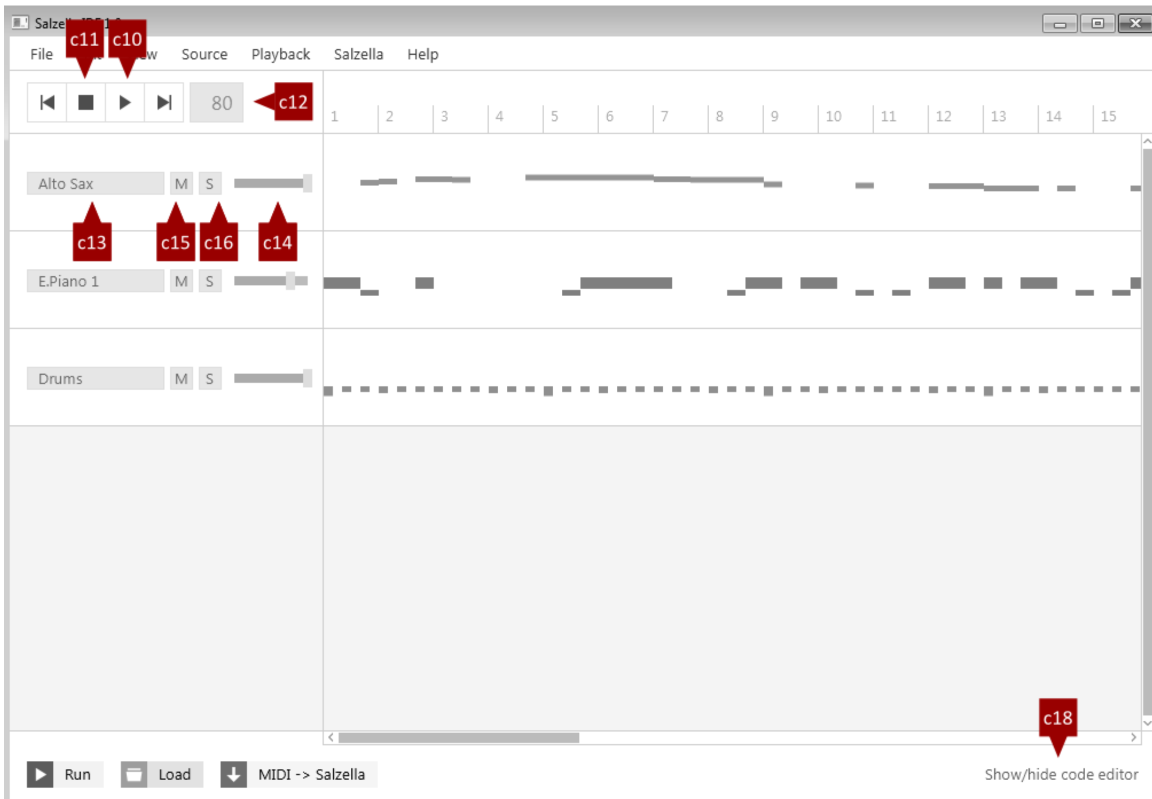


Figure 5.4: Playback of musical pieces

5.1.4 MIDI export/import

Figure 5.5 uses labels $c19$-$c20$ to mark controls relevant to exporting/importing musical pieces to/from MIDI files. To import a MIDI file, click on button $c19$ and use the dialog window to locate the MIDI file to be imported. To export contents of the currently loaded musical piece to a MIDI file, click on button $c20$ and use the dialog window to specify the output file.

5.1.4.1 Functional requirements coverage (4/5)

Controls $c19$-$c20$ are related to the following functional requirements:

$r7$ - Save musical pieces to files

$r8$ - Load musical pieces from files

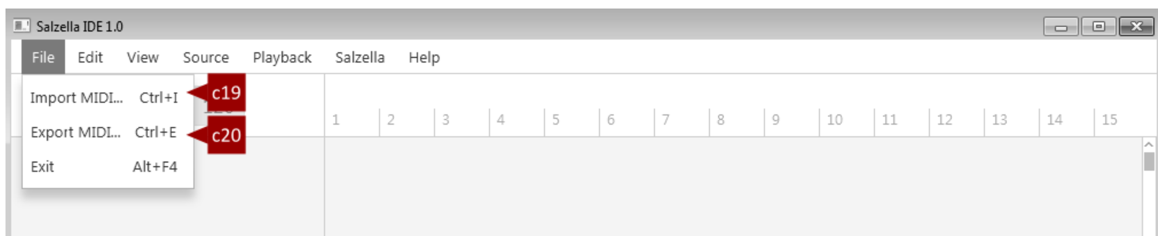


Figure 5.5: MIDI export/import

5.1.5 Performing undo/redo

Figure 5.6 uses labels $c21$-$c22$ to mark controls relevant to undoing/redoing changes made to musical pieces. Any changes made to properties of the currently loaded musical piece can be undone using $c21$ and redone using $c22$.

5.1.5.1 Functional requirements coverage (5/5)

Controls $c21$-$c22$ are related to the following functional requirements:

$r9$ - Perform undo/redo operations over $r1$, $r4$ and $r5a$-$r5e$

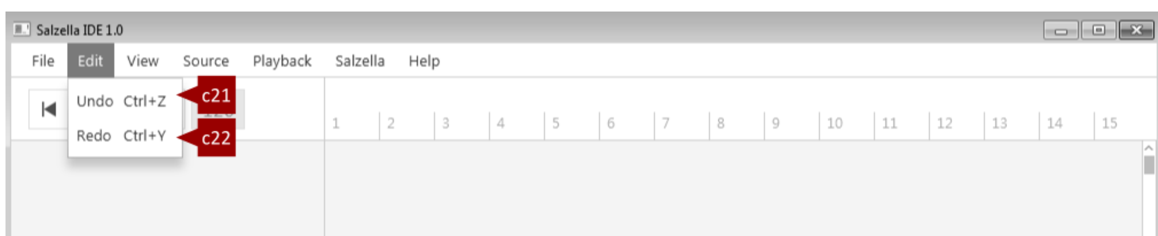


Figure 5.6: Performing undo/redo

5.2 Realization

5.2.1 Architecture

Salzella development environment is built on top of the MVC architecture [17]. Communication between layers is handled by standard GoF design patterns [18]. Undoable changes in Model are performed using the command pattern and Model-View synchronization is achieved by means of the observer pattern. The implementation is organized in seven packages as described below.

5.2.1.1 Package overviews

Salzella development environment is built on top of the MVC architecture. The actual implementation is organized as follows:

`cz.stepanvolf.salzie`
- Initialization of the application.

`cz.stepanvolf.salzie.model`
- Model of the application. Instances of classes which reside in this package represent the application state. For example, instance of the `Project` class stores information about the currently loaded musical piece and offers methods such as `setTempo()` which can be used to change its properties. Instance of the `Executor` class stores undo/redo lists and is responsible for the actual execution of these commands. Note that this package also contains XML files which store information about presets and list of available plugins. For complete overview of classes and configuration files which reside in this package, refer to javadoc documentation which can be found on the enclosed CD, see Appendix D.

`cz.stepanvolf.commands`
- Commands used to perform undoable changes on model.

`cz.stepanvolf.controller`
- Controller of the application. Upon receiving information about some user action, it either modifies model by means of creating and executing a command or requests some service such as playback of a musical piece. Controller of the application is implemented in a single class called `Controller`. The instance of this class is available to classes from the View layer by means of the singleton pattern. For each possible user action, a method exits in `Controller` which is bound to be invoked upon occurrence of the action. The names of these methods should be self explanatory. Here are a few examples: `tempoChanged(int tempo)`, `instrumentChanged(int track, int instrument)`, `undoPressed()`.

`cz.stepanvolf.observers`
- Interfaces to be implemented by observers of the application state.

`cz.stepanvolf.view`
- Implementation of the user interface.

`cz.stepanvolf.images`
- Images of icons used in the user interface.

5.2.2 Implementation notes

The development environment was implemented using the Java programming language. Source codes were version controlled using Git. Both the source code and javadoc documentation can be found on the enclosed CD, see Appendix D. Note that the development environment uses a BSD licensed [19] open source implementation of rich text area developed by Thomas Munsel [20]. The compiled versions of external libraries as well as executable version of the development environment can be found on the enclosed CD.

Chapter 6

Conclusion

6.1 Evaluation

The evaluation of the results of this thesis will be divided into three separate parts. First, the chosen methodology will be evaluated. Next, the algorithms responsible for the actual music generation will be evaluated. And finally, advantages and disadvantages of the created language will be discussed.

6.1.1 Methodology

The goal of this thesis was to create a declarative language for music generation which would allow its users to encapsulate the essence of an already existing musical piece in form of a program. Executing this program should result in producing a musical piece similar to the original. To formally define whether one musical piece is similar to another, a binary relation over all musical pieces had to be defined. This was achieved by creating a simple function which realizes this relation. This function was formally presented as an instance of a so called similarity criterion and its creation was described in 2.5. The formal definition of similarity criterion didn't require the implied similarity relation to be neither symmetric nor transitive which helped in terms of computational feasibility. Since the function is in practice used to produce one similar musical piece at the time, the user of this function has no chance of detecting the relation is not transitive.

As mentioned many times, there is no universally correct definition of what makes two musical pieces similar. The created similarity criterion is therefore not the only possible definition of the similarity relation. It is a prototype with very limited capabilities. However, it proved to be useful when designing the actual language which was created as part of a so called extended similarity criterion as defined in 1.1.3.3. Reflecting on the fact that there may be many definitions of what makes two musical pieces similar, the created language was designed to be capable of absorbing functionality of various similarity criterions. This resulted in the most important structural feature of the proposed language which is extensibility.

6.1.2 Algorithms

Three algorithms which deal with the actual music generation were created as part of the similarity criterion. The most sophisticated of these algorithms was algorithm for melody harmonization described in 2.2. This algorithm uses a set of scoring matrices to encapsulate relations among tones, chords and musical keys to find optimal solution in exponentially big solution space. Existence of several different sets of these scoring matrices is presumed. These sets can be designed to address different musical genres, or even compositional styles of particular composers. Automated creation of these matrices based on data sets of already existing musical pieces was not solved as part of this thesis but it would definitely be the next logical step. A modified version of this algorithm capable of performing harmony analysis of an existing musical piece was used in the created similarity criterion.

The melody generating algorithm described 2.3 is based on idea of generating sequences of shorter melody segments which satisfy a simple local constraint. This technique proved to be useful when generating variations on existing melodies. However, used as a general purpose melody generating algorithm, this algorithm would have many limitations. For example, it doesn't allow its users to say anything about structure of the generated melody in terms of repetitions and variations.

The algorithm responsible for percussion generation which was described in 2.4 was created mainly to illustrate the genre independence capabilities of the created language. The algorithm is capable of generating simple rhythm loops. The explicit positioning of snare hits within the rhythm and randomized positioning of kick and open hi-hat hits on top of semi-regular grid of closed hi-hat hits seem to provide more than enough control to create genre specific rhythms.

6.1.3 Language

Intended users of the created language are creators of music generating software. Evaluation of the language will be therefore done through answering the following question: Why should developers of music generating software use Salzella as internal engine?

First of all, it has to be said that the capabilities of Salzella are directly proportional to quality of its extensions. The three extensions which were created as part of this thesis are very simple prototypes. The creators of music generating tools who will decide to use Salzella are, however, free to create their own extensions and thus modify the functionality of Salzella. In fact, they can even completely bypass the built-in support for describing the overall structure of musical pieces.

Creating a Salzella extension is a simple matter of implementing an interface with a single method. This method will be responsible for modifying contents of a musical piece. Salzella defines its own data types to represent the musical contents and provides methods which can handle conversion between these data types and standard MIDI sequences. This means that even already existing implementations of algorithms which use standard MIDI sequences can be integrated into Salzella.

Taking advantage of the fact that Salzella programs are presumed to be generated rather than written by hand, the language does not provide explicit support for abstract structures

like chords, scales and keys. Instead, these structures can be manually defined within individual programs. This low level approach was the most important decision made during creation of the language. In retrospect, this approach had the following advantages and disadvantages:

Advantages The most obvious advantage of adopting the low level approach was overall simplification of the language. Possibility of manual definition of arbitrary scales/chords was achieved by providing specification of a single type of matrix, the surface matrix. Note that the concept of surface matrices plays the key role in satisfying the music genre independence requirement. By even the slightest modification of program's surface matrix, contents of the outputted musical piece can change dramatically. For example, program for generating rock music, see B.1, uses the surface matrix to define a chord progression made of power chords and declares usage of pentatonic scale. Program for generating folk music, see B.4, uses the same data structure to define harmony based on major/minor triads and declares usage of diatonic scale.

Disadvantages The biggest disadvantage of the low level approach is, of course, delegating the responsibility of defining the high level structures to the user. Adopting the low level approach also implies growth of Salzella programs in terms of numbers of lines. Salzella programs can be as long as tens of thousands of lines. But since ease of manual creation of Salzella programs was not a priority, neither of these disadvantages presented a problem.

6.2 Future work

6.2.1 Verse and chorus support

The current version of Salzella works best when dealing with musical ideas few bars long. Even though it would be possible to create an extension which would allow Salzella users to work with concepts like verses and choruses, adding explicit support for defining custom meaning of manually selected time windows should be considered.

6.2.2 Surface matrix convention

The twelve-integer surface matrix convention doesn't seem to be sufficient when dealing with more complex music. One possible course of action could be using twenty four integers per row. This would allow users to store information about scales and chords separately. Going even further and using thirty six integers per row would make it possible to use different scales based on direction of melodies. This would be particularly useful when dealing with melodic minor scales. No matter what approach will be chosen, limiting the size of surface matrix rows is unnecessary and should be definitely removed from Salzella specification.

6.2.3 Manual creation of surface matrices

Even though Salzella programs are presumed to be generated rather than written by hand, developers of Salzella extension will often want to create surface matrices manually to test

their algorithms. Manual creation of surface matrices is a tedious and error prone task. Once surface matrix convention is stable enough, a surface matrix generation tool should be added to the development environment to make manual creation of surface matrices easier.

Bibliography

[1] "MIDI Specification." MIDI Manufacturers Association. Accessed December 23, 2015. http://midi.org/.

[2] Biedrzycki, Maciej. "Can Computers Create Music?" CgMusic. May 19, 2008. Accessed December 23, 2015. http://codeminion.com/blogs/maciek/2008/05/cgmusic-computers-create-music/.

[3] "User-Customized Jazz Improvisation Generation." Jazzerbot. Accessed December 23, 2015. https://code.google.com/p/jazzerbot/.

[4] "An Experiment in a New Kind of Music." WolframTones. Accessed December 23, 2015. http://tones.wolfram.com/.

[5] "Music Programming for Java™ and JVM Languages." JFugue. Accessed December 23, 2015. http://www.jfugue.org/.

[6] "Computer Music Composition in Java." JMusic. Accessed December 23, 2015. http://explodingart.com/jmusic/.

[7] Volf, Stepan. Music Generator (M-Architect). Bachelor's thesis, Czech Technical University in Prague, 2012.

[8] Pratchett, Terry. Maskerade: A Novel of Discworld Series. New York: HarperPrism, 1997.

[9] Yogev, Noam, and Alexander Lerch. "A System for Automatic Audio Harmonization." Proceedings of the VdT International Convention (25. Tonmeistertagung), 2008.

[10] Phon-Amnuaisuk, Somnuk, and Geraint A. Wiggins. "The Four-part Harmonisation Problem: A Comparison between Genetic Algorithms and a Rule-Based System." Proceedings of the AISB '99 Symposium on Musical Creativity, 1999, 28-34.

[11] Pachet, François, and Pierre Roy. "Mixing Constraints and Objects: A Case Study in Automatic Harmonization." Prentice-Hall, TOOLS Europe, 1995, 119-26.

[12] Kathiresan, Thayabaran. Automatic Melody Generation. Master's thesis, KTH Royal Institute of Technology, 2015.

[13] "American National Standard for Information Systems — Coded Character Sets — 7-Bit American National Standard Code for Information Interchange (7-Bit ASCII), ANSI X3.4-1986". American National Standards Institute, March 26, 1986.

[14] "Documentation of java.util.regex.Pattern." Java Platform SE 8. Accessed May 15, 2016. https://docs.oracle.com/javase/8/docs/api/java/util/regex/Pattern.html

[15] Backus, John Warner. "The Syntax and Semantics of the Proposed International Algebraic Language of the Zurich ACM-GAMM Conference." Proceedings of the International Conference on Information Processing, 1959, 125-32.

[16] "ISO/IEC 14977: Information Technology - Syntactic Metalanguage - Extended BNF." International Organization for Standardization, December 15, 1996.

[17] Fowler, Martin. "GUI Architectures." July 18, 2006. Accessed May 15, 2016. http://martinfowler.com/eaaDev/uiArchs.html.

[18] Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Reading, MA: Addison-Wesley, 1995.

[19] "The BSD 2-Clause License." Open Source Initiative. Accessed May 15, 2016. https://opensource.org/licenses/BSD-2-Clause.

[20] Mikula, Tomas. "RichTextFX: Rich-text area for JavaFX." GitHub. Accessed May 15, 2016. https://github.com/TomasMikula/RichTextFX.

Appendix A

Lengthy definitions

A.1 Instrument mapping

A.1.1 Instrument mapping

Let $Identifier = \{0, 1, 2, ..., 128\}$. Let $Instrument$ be a set of string representations of types of musical instruments. Instrument mapping is a function $\iota : Identifier \to Instrument$ which assigns a type of musical instrument to each integer identifier. Below is a full enumerative definition of instrument mapping.

0 → Acoustic Grand Piano
1 → Bright Acoustic Piano
2 → Electric Grand Piano
3 → Honky-tonk Piano
4 → Rhodes Piano
5 → Chorused Piano
6 → Harpsichord
7 → Clavinet
8 → Celesta
9 → Glockenspiel
10 → Music Box
11 → Vibraphone
12 → Marimba
13 → Xylophone
14 → Tubular Bells
15 → Dulcimer
16 → Drawbar Organ
17 → Percussive Organ
18 → Rock Organ
19 → Church Organ
20 → Reed Organ
21 → Accordion
22 → Harmonica

23 → Tango Accordion
24 → Acoustic Guitar (nylon)
25 → Acoustic Guitar (steel)
26 → Electric Guitar (jazz)
27 → Electric Guitar (clean)
28 → Electric Guitar (muted)
29 → Overdriven Guitar
30 → Distortion Guitar
31 → Guitar Harmonics
32 → Acoustic Bass
33 → Electric Bass (finger)
34 → Electric Bass (pick)
35 → Fretless Bass
36 → Slap Bass 1
37 → Slap Bass 2
38 → Synth Bass 1
39 → Synth Bass 2
40 → Violin
41 → Viola
42 → Cello
43 → Contrabass
44 → Tremolo Strings
45 → Pizzicato Strings
46 → Orchestral Harp
47 → Timpani
48 → Strings
49 → Slow Strings
50 → Synth Strings 1
51 → Synth Strings 2
52 → Choir Aahs
53 → Voice Oohs
54 → Synth Choir
55 → Orchestra Hit
56 → Trumpet
57 → Trombone
58 → Tuba
59 → Muted Trumpet
60 → French Horn
61 → Brass Section
62 → Synth Brass 1
63 → Synth Brass 2
64 → Soprano Sax
65 → Alto Sax
66 → Tenor Sax
67 → Baritone Sax

```
68 → Oboe
69 → English Horn
70 → Bassoon
71 → Clarinet
72 → Piccolo
73 → Flute
74 → Recorder
75 → Pan Flute
76 → Blown bottle
77 → Shakuhachi
78 → Whistle
79 → Ocarina
80 → Square Wave
81 → Saw Wave
82 → Synth Calliope
83 → Chiffer Lead
84 → Charang
85 → Solo Vox
86 → 5th Saw Wave
87 → Bass & Lead
88 → Fantasia
89 → Warm Pad
90 → Polysynth
91 → Space Voice
92 → Bowed Glass
93 → Metal Pad
94 → Halo Pad
95 → Sweep Pad
96 → Ice Rain
97 → Soundtrack
98 → Crystal
99 → Atmosphere
100 → Brightness
101 → Goblin
102 → Echo Drops
103 → Star Theme
104 → Sitar
105 → Banjo
106 → Shamisen
107 → Koto
108 → Kalimba
109 → Bagpipe
110 → Fiddle
111 → Shanai
112 → Tinkle Bell
```

113 → Agogo
114 → Steel Drums
115 → Woodblock
116 → Taiko
117 → Melodic Tom
118 → Synth Drum
119 → Reverse Cymbal
120 → Guitar Fret Noise
121 → Breath Noise
122 → Seashore
123 → Bird Tweet
124 → Telephone Ring
125 → Helicopter
126 → Applause
127 → Gun Shot
128 → Percussion

A.2 Percussion mapping

A.2.1 Percussion mapping

Let $Identifier = \{35, 36, 37, ..., 81\}$. Let $Percussion$ be a set of string representations of types of percussive sounds. Percussion mapping is a function $\pi : Identifier \rightarrow Percussion$ which assigns a type of percussive sound to each integer identifier. Below is a full enumerative definition of percussion mapping.

35 → Acoustic Bass Drum
36 → Bass Drum
37 → Side Stick
38 → Acoustic Snare
39 → Hand Clap
40 → Electric Snare
41 → Low Floor Tom
42 → Closed Hi Hat
43 → High Floor Tom
44 → Pedal Hi-Hat
45 → Low Tom
46 → Open Hi-Hat
47 → Low-Mid Tom
48 → Hi Mid Tom
49 → Crash Cymbal 1
50 → High Tom
51 → Ride Cymbal 1
52 → Chinese Cymbal
53 → Ride Bell

54 → Tambourine
55 → Splash Cymbal
56 → Cowbell
57 → Crash Cymbal 2
58 → Vibraslap
59 → Ride Cymbal 2
60 → Hi Bongo
61 → Low Bongo
62 → Mute Hi Conga
63 → Open Hi Conga
64 → Low Conga
65 → High Timbale
66 → Low Timbale
67 → High Agogo
68 → Low Agogo
69 → Cabasa
70 → Maracas
71 → Short Whistle
72 → Long Whistle
73 → Short Guiro
74 → Long Guiro
75 → Claves
76 → Hi Wood Block
77 → Low Wood Block
78 → Mute Cuica
79 → Open Cuica
80 → Mute Triangle
81 → Open Triangle

A.3 Playback

A.3.1 Playback

Let m be an instance of musical piece. To play back m means to produce a sound for each $e \in m.Events$. Time at which the production of this sound should start can be obtained by calculating by $(e.start/m.resolution)(60/m.tempo)$. Time at which the production of this sound should terminate can be calculated using the same formula with $e.start$ substituted with $e.end$. Value obtained by the time formula should be interpreted as number of seconds from initiation of playback of m. The relative volume of the produced sound can be calculated by $(e.velocity/127)$. Value obtained by the volume formula should be interpreted as relative volume where 0 means silent and 1 means as loud as possible. If $m.\rho(e).instrument < 128$, then the produced sound should be a tone with frequency implied by $e.pitch$ and instrument used to produce this tone should be $\iota(m.\rho(e).instrument)$. If $m.\rho(e).instrument = 128$ the produced

sound should be a percussion hit $\pi(m.\rho(e).instrument)$.

A.4 Integer literals

A.4.1 Integer literals

```
<d0> ::= "0"
<d1> ::= "0" | "1"
<d2> ::= "0" | "1" | "2"
<d3> ::= "0" | "1" | "2" | "3"
<d4> ::= "0" | "1" | "2" | "3" | "4"
<d5> ::= "0" | "1" | "2" | "3" | "4" | "5"
<d6> ::= "0" | "1" | "2" | "3" | "4" | "5" | "6"
<d7> ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7"
<d8> ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8"
<d9> ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
<positive> ::= "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"

<integer:1-INF> ::=
  <positive> [<d9>] [<d9>] [<d9>] [<d9>] [<d9>] [<d9>] [<d9>] [<d9>] |
                    "1" <d9> <d9> <d9> <d9> <d9> <d9> <d9> <d9> <d9> |
                    "2" <d0> <d9> <d9> <d9> <d9> <d9> <d9> <d9> <d9> |
                    "2"  "1" <d3> <d9> <d9> <d9> <d9> <d9> <d9> <d9> |
                    "2"  "1"  "4" <d6> <d9> <d9> <d9> <d9> <d9> <d9> |
                    "2"  "1"  "4"  "7" <d3> <d9> <d9> <d9> <d9> <d9> |
                    "2"  "1"  "4"  "7"  "4" <d7> <d9> <d9> <d9> <d9> |
                    "2"  "1"  "4"  "7"  "4"  "8" <d2> <d9> <d9> <d9> |
                    "2"  "1"  "4"  "7"  "4"  "8"  "3" <d5> <d9> <d9> |
                    "2"  "1"  "4"  "7"  "4"  "8"  "3"  "6" <d3> <d9> |
                    "2"  "1"  "4"  "7"  "4"  "8"  "3"  "6"  "4" <d7>
<integer:0-127> ::= <d9> | <positive> <d9> | "1" <d1> <d9> | "1"  "2" <d7>
<integer:0-128> ::= <integer:0-127> | "128"
<integer:0-INF> ::= <integer:1-INF> | "0"
```

A.5 Literal mappings

A.5.1 Pitch literal mapping

c0 $\rightarrow 0$
c#0 $\rightarrow 1$
d0 $\rightarrow 2$
d#0 $\rightarrow 3$

```
e0  → 4
f0  → 5
f#0 → 6
g0  → 7
g#0 → 8
a0  → 9
a#0 → 10
b0  → 11
c1  → 12
c#1 → 13
d1  → 14
d#1 → 15
e1  → 16
f1  → 17
f#1 → 18
g1  → 19
g#1 → 20
a1  → 21
a#1 → 22
b1  → 23
c2  → 24
c#2 → 25
d2  → 26
d#2 → 27
e2  → 28
f2  → 29
f#2 → 30
g2  → 31
g#2 → 32
a2  → 33
a#2 → 34
b2  → 35
c3  → 36
c#3 → 37
d3  → 38
d#3 → 39
e3  → 40
f3  → 41
f#3 → 42
g3  → 43
g#3 → 44
a3  → 45
a#3 → 46
b3  → 47
c4  → 48
```

`c#4` → 49
`d4` → 50
`d#4` → 51
`e4` → 52
`f4` → 53
`f#4` → 54
`g4` → 55
`g#4` → 56
`a4` → 57
`a#4` → 58
`b4` → 59
`c5` → 60
`c#5` → 61
`d5` → 62
`d#5` → 63
`e5` → 64
`f5` → 65
`f#5` → 66
`g5` → 67
`g#5` → 68
`a5` → 69
`a#5` → 70
`b5` → 71
`c6` → 72
`c#6` → 73
`d6` → 74
`d#6` → 75
`e6` → 76
`f6` → 77
`f#6` → 78
`g6` → 79
`g#6` → 80
`a6` → 81
`a#6` → 82
`b6` → 83
`c7` → 84
`c#7` → 85
`d7` → 86
`d#7` → 87
`e7` → 88
`f7` → 89
`f#7` → 90
`g7` → 91
`g#7` → 92
`a7` → 93

a#7 → 94
b7 → 95
c8 → 96
c#8 → 97
d8 → 98
d#8 → 99
e8 → 100
f8 → 101
f#8 → 102
g8 → 103
g#8 → 104
a8 → 105
a#8 → 106
b8 → 107
c9 → 108
c#9 → 109
d9 → 110
d#9 → 111
e9 → 112
f9 → 113
f#9 → 114
g9 → 115
g#9 → 116
a9 → 117
a#9 → 118
b9 → 119
c10 → 120
c#10 → 121
d10 → 122
d#10 → 123
e10 → 124
f10 → 125
f#10 → 126
g10 → 127

A.5.2 Duration base mapping

1 → 384
2 → 192
4 → 96
8 → 48
16 → 24
32 → 12
64 → 6

`1t` $\rightarrow 256$
`2t` $\rightarrow 128$
`4t` $\rightarrow 64$
`8t` $\rightarrow 32$
`16t` $\rightarrow 16$
`32t` $\rightarrow 8$
`64t` $\rightarrow 4$

`1d` $\rightarrow 576$
`2d` $\rightarrow 288$
`4d` $\rightarrow 144$
`8d` $\rightarrow 72$
`16d` $\rightarrow 36$
`32d` $\rightarrow 18$
`64d` $\rightarrow 9$

Appendix B

Example programs

B.1 Program 1 - Rock

B.1.0.1 Program 1 - Rock

```
     tempo: 120;
   surface: 1x1 0 0 1 0 3 0 0 1 0 1 0 2,
            1x1 0 0 3 0 1 0 0 1 0 2 0 1,
            1x1 3 0 1 0 1 0 0 2 0 1 0 1,
            1x1 1 0 1 0 1 0 2 1 0 1 0 3,
            1x1 0 0 1 0 3 0 0 1 0 1 0 2,
            1x1 0 0 3 0 1 0 0 1 0 2 0 1,
            1x1 3 0 1 0 1 0 0 2 0 1 0 1,
            1x1 1 0 1 0 1 0 2 1 0 1 0 3;
 signature: 8 4 1x4;
    tracks: guitar 30 105 false false,
            organ  18  95 false false,
            muted  28  70 false false,
            drums 128 120 false false;
            ---
    source: cz.stepanvolf.salzella.plugin.GeneralMelody;
     input: guitar;
     start: 2x1;
  duration: 2x1;
      grid: 1x8;
 structure: 0.3 0.8,
            UP DOWN UP UP;
     chord: 0.5;
     relax: 0.2;
       min: 0;
       max: 1;
       low: e4;
```

```
      high: e5;
            ---
    source: cz.stepanvolf.salzella.plugin.GeneralMelody;
     input: guitar;
     start: 4x1;
  duration: 4x1;
      grid: 1x8;
 structure: 0.3 0.5 0.6,
            UP DOWN STEADY STEADY UP;
     chord: 0.7;
     relax: 0.8;
       min: 0;
       max: 2;
       low: e3;
      high: e6;
            ---
    source: cz.stepanvolf.salzella.plugin.SimpleDrums;
     input: drums;
     start: 0x1;
  duration: 8x1;
      loop: 1x1;
     hihat: 1x8;
     snare: 1x4 3x4;
      kick: 0.3;
     crash: 0.2;
   variety: 0.1;
            ---
    source: cz.stepanvolf.salzella.plugin.RhythmGuitar;
     input: organ drums;
     start: 0x1;
  duration: 8x1;
      mode: RHYTHM;
      grid: 1x8;
            ---
    source: cz.stepanvolf.salzella.plugin.RhythmGuitar;
     input: muted drums;
     start: 0x1;
  duration: 8x1;
      mode: RHYTHM;
      grid: 1x8;
```

B.2 Program 2 - Blues

B.2.0.2 Program 2 - Blues

```
     tempo: 80;
   surface: 4x1 0 0 2 0 3 0 0 1 2 1 0 2,
            2x1 0 2 1 0 2 0 0 2 0 3 0 1,
            2x1 0 0 2 0 3 0 0 1 2 1 0 2,
            1x1 0 0 1 2 1 0 2 1 0 2 0 3,
            1x1 0 2 1 0 2 0 0 2 0 3 0 1,
            1x1 0 0 2 0 3 0 0 1 2 1 0 2,
            1x1 0 0 1 2 1 0 2 1 0 2 0 3;
 signature: 12 4 1x4;
    tracks: sax    65 120 false false,
            piano   4  90 false false,
            drums 128 120 false false;
            ---
    source: cz.stepanvolf.salzella.plugin.GeneralMelody;
     input: sax;
     start: 0x1;
  duration: 12x1;
      grid: 1x8t;
 structure: 0.3 0.5 0.5 0.7,
            UP DOWN DOWN UP STEADY STEADY STEADY UP UP;
     chord: 0.4;
     relax: 0.2;
       min: 0;
       max: 3;
       low: e4;
      high: b5;
            ---
    source: cz.stepanvolf.salzella.plugin.SimpleDrums;
     input: drums;
     start: 0x1;
  duration: 12x1;
      loop: 1x1;
     hihat: 1x8t;
     snare: 1x4 3x4;
      kick: 0.3;
     crash: 0.0;
   variety: 0.0;
            ---
    source: cz.stepanvolf.salzella.plugin.RhythmGuitar;
     input: piano;
     start: 0x1;
```

```
  duration: 12x1;
      mode: RHYTHM;
      grid: 1x8t;
```

B.3 Program 3 - Jazz

B.3.0.3 Program 3 - Jazz

```
     tempo: 60;
   surface: 1x1 0 0 2 0 0 0 2 3 0 0 0 2,
            1x1 3 0 0 0 2 0 0 2 0 0 0 2,
            1x1 2 0 0 0 2 0 0 2 0 3 0 0,
            1x1 0 0 0 2 0 0 2 0 3 0 0 2;
 signature: 4 4 1x4;
    tracks: guitar 26 100 false false,
            piano  26  90 false false,
            bass   32 100 false false,
            drums 128 120 false false;
            ---
    source: cz.stepanvolf.salzella.plugin.GeneralMelody;
     input: guitar;
     start: 0x1;
  duration: 2x1;
      grid: 1x8t;
 structure: 0.3 0.5,
            UP DOWN STEADY;
     chord: 0.2;
     relax: 0.3;
       min: 0;
       max: 3;
       low: c4;
      high: e6;
            ---
    source: cz.stepanvolf.salzella.plugin.GeneralMelody;
     input: guitar;
     start: 2x1;
  duration: 2x1;
      grid: 1x4;
 structure: 0.8 0.5,
            DOWN UP UP;
     chord: 0.5;
     relax: 0.8;
       min: 0;
       max: 2;
       low: c4;
```

```
       high: e6;
             ---
     source: cz.stepanvolf.salzella.plugin.SimpleDrums;
      input: drums;
      start: 0x1;
   duration: 4x1;
       loop: 1x1;
      hihat: 1x16;
      snare: 1x4 1x2 3x4;
       kick: 0.3;
      crash: 0.2;
    variety: 0.8;
             ---
     source: cz.stepanvolf.salzella.plugin.RhythmGuitar;
      input: piano;
      start: 0x1;
   duration: 12x1;
       mode: STRUM;
       grid: 1x4t;
             ---
     source: cz.stepanvolf.salzella.plugin.GeneralMelody;
      input: bass;
      start: 0x1;
   duration: 4x1;
       grid: 1x4;
  structure: 0.3,
             STEADY;
      chord: 1.0;
      relax: 1.0;
        min: 0;
        max: 12;
        low: e2;
       high: e3;
```

B.4 Program 4 - Folk

B.4.0.4 Program 4 - Folk

```
      tempo: 120;
    surface: 1x1 1 0 2 0 1 0 1 3 0 1 0 2,
             1x1 1 0 1 0 3 0 1 2 0 1 0 2,
             1x1 3 0 1 0 2 0 1 2 0 1 0 1,
             1x1 1 0 2 0 1 0 1 3 0 1 0 2,
             1x1 1 0 2 0 1 0 1 3 0 1 0 2,
             1x1 1 0 1 0 3 0 1 2 0 1 0 2,
```

```
             2x1 1 0 3 0 1 0 2 1 0 2 0 1;
  signature: 8 4 1x4;
     tracks: harmonica 22 100 false false,
             guitar    25 100 false false;
             ---
     source: cz.stepanvolf.salzella.plugin.GeneralMelody;
      input: harmonica;
      start: 0x1;
   duration: 4x1;
       grid: 1x8;
  structure: 0.7 0.5,
             UP DOWN UP STEADY;
      chord: 1.0;
      relax: 0.5;
        min: 0;
        max: 2;
        low: g3;
       high: g5;
             ---
     source: cz.stepanvolf.salzella.plugin.GeneralMelody;
      input: harmonica;
      start: 4x1;
   duration: 4x1;
       grid: 1x4;
  structure: 0.9 0.3 0.5 0.9,
             DOWN DOWN UP UP;
      chord: 1.0;
      relax: 0.5;
        min: 0;
        max: 3;
        low: g4;
       high: g6;
             ---
     source: cz.stepanvolf.salzella.plugin.RhythmGuitar;
      input: guitar;
      start: 0x1;
   duration: 8x1;
       mode: RHYTHM;
       grid: 1x16;
```

B.5 Program 5 - Classical

B.5.0.5 Program 5 - Classical

```
    tempo: 120;
  surface: 1x1 1 0 1 0 3 0 1 2 0 1 0 2,
           1x1 1 0 2 0 1 0 2 1 0 1 0 3,
           1x1 1 0 1 0 3 0 1 2 0 1 0 2,
           1x1 1 0 2 0 1 0 2 1 0 1 0 3,
           1x1 3 0 1 0 2 0 1 2 0 1 0 1,
           1x1 1 0 3 0 1 0 2 1 0 2 0 1,
           2x1 1 0 1 0 3 0 1 2 0 1 0 2;
signature: 8 4 1x4;
   tracks: violin     40 105 false false,
           cello      42  95 false false,
           contrabass 43  95 false false;
           ---
   source: cz.stepanvolf.salzella.plugin.GeneralMelody;
    input: violin;
    start: 2x1;
 duration: 6x1;
     grid: 1x8;
structure: 0.2,
           STEADY;
    chord: 0.9;
    relax: 1.0;
      min: 0;
      max: 3;
      low: e5;
     high: e6;
           ---
   source: cz.stepanvolf.salzella.plugin.GeneralMelody;
    input: cello;
    start: 1x1;
 duration: 7x1;
     grid: 1x4;
structure: 0.3,
           STEADY;
    chord: 0.2;
    relax: 1.0;
      min: 0;
      max: 12;
      low: e3;
     high: e4;
           ---
```

```
      source: cz.stepanvolf.salzella.plugin.GeneralMelody;
       input: contrabass;
       start: 0x1;
    duration: 8x1;
        grid: 1x2;
   structure: 1.0,
              STEADY;
       chord: 1.0;
       relax: 1.0;
         min: 0;
         max: 12;
         low: e2;
        high: e3;
```

Appendix C

UML diagrams

C.1 Salzella object model

.

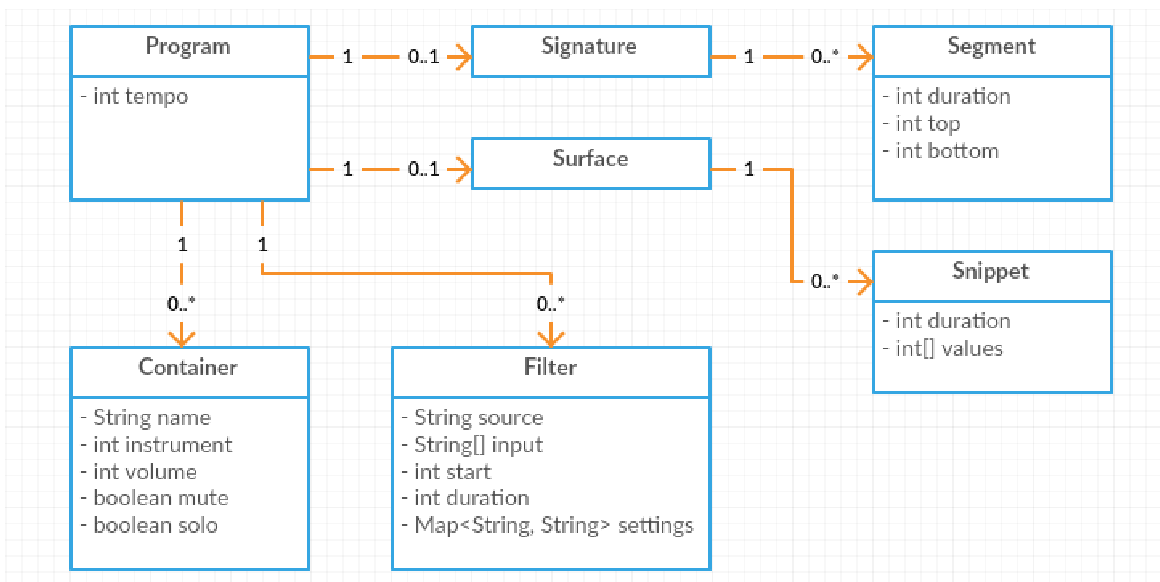


Figure C.1: Salzella object model

C.2 Lightweight MIDI entities

.

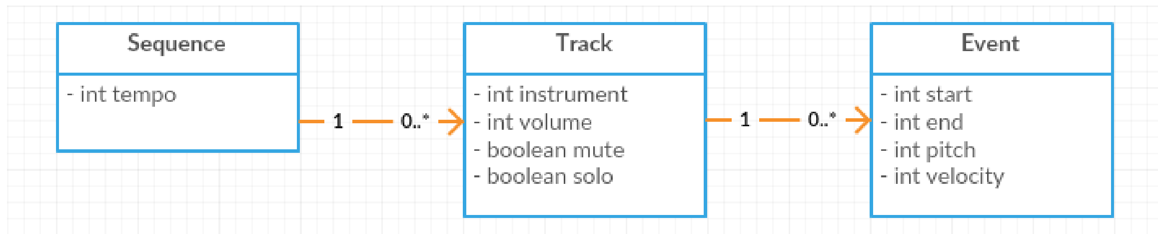


Figure C.2: Lightweight MIDI entities

Appendix D

Contents of the enclosed CD

```
├── javadoc
│   ├── converter-javadoc.zip
│   ├── ide-javadoc.zip
│   ├── plugins-javadoc.zip
│   └── salzella-javadoc.zip
├── misc
│   └── older-documentation.pdf
├── samples
│   ├── greensleeves.mid
│   ├── jethro.mid
│   └── scarborough.mid
├── source
│   ├── converter-source.zip
│   ├── ide-dependencies.zip
│   ├── ide-source.zip
│   ├── plugins-source.zip
│   └── salzella-source.zip
└── ide.jar
```