Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

Extending MIDI encoding with metadata from harmonic analysis of classical music

Bachelor Thesis

2024
Jakub Tarhovický

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

Extending MIDI encoding with metadata from harmonic analysis of classical music

Bachelor Thesis

Study Programme:	Computer Science
Field of Study:	Computer Science
Department:	Department of Computer Science
Supervisor:	RNDr. Andrej Ferko, PhD

Bratislava, 2024
Jakub Tarhovický

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Jakub Tarhovický
Študijný program: informatika (Jednoodborové štúdium, bakalársky I. st., denná forma)
Študijný odbor: informatika
Typ záverečnej práce: bakalárska
Jazyk záverečnej práce: anglický
Sekundárny jazyk: slovenský

Názov: Extending MIDI encoding with metadata from harmonic analysis of classical music
Rozšírenie kódovania MIDI o metadáta z harmonickej analýzy klasickej hudby

Anotácia: Textové výstupy automatickej analýzy klasických diel vo forme časových radov na báze MIDI vyžadujú transformáciu kvôli enharmonickým tónom a iným problémom. Práca porovnáva publikované transkripčné kódovania, navrhuje efektívnu modifikáciu a implementáciu.

Cieľ: Prehľad problematiky. Špecifikácia riešenia. Implementácia a vyhodnotenie výsledkov.

Literatúra: SELFRIDGE-FIELD, E. Beyond MIDI. The Handbook of Musical Codes. MIT Press. ISBN: 9780262193948.
FERKOVÁ, E. - ADAMOVÁ, S. - ŠUKOLA, M. - URBANCOVÁ, H. 2021. Computer search tool for harmonic structures and progressions in MIDI files and possible applications. Clavibus Unitis 2020, Vol. 9. No. 3. [online] https://acecs.cz/media/cu_2020_09_03_ferkova.pdf.

Kľúčové slová: Kódovanie hudby, MIDI, harmonická analýza, rozšírenie MIDI, hudba, kódovanie

Vedúci: doc. RNDr. Andrej Ferko, PhD.
Katedra: FMFI.KAG - Katedra algebry a geometrie
Vedúci katedry: doc. RNDr. Pavel Chalmovianský, PhD.

Dátum zadania: 14.09.2022

Dátum schválenia: 18.10.2023 doc. RNDr. Dana Pardubská, CSc.
garant študijného programu

.....
študent

.....
vedúci práce

Comenius University Bratislava
Faculty of Mathematics, Physics and Informatics

THESIS ASSIGNMENT

Name and Surname:	Jakub Tarhovický
Study programme:	Computer Science (Single degree study, bachelor I. deg., full time form)
Field of Study:	Computer Science
Type of Thesis:	Bachelor´s thesis
Language of Thesis:	English
Secondary language:	Slovak

Title: Extending MIDI encoding with metadata from harmonic analysis of classical music

Annotation: The text outputs of the automatic analysis of classical works in the form of MIDI-based time series require transformation due to enharmonic tones and other phenomena. The project compares published transcription encodings beyond MIDI, proposes an effective modification and implementation.

Aim: Previous work overview. Software specification. Implementation and evaluation of the results.

Literature: SELFRIDGE-FIELD, E. Beyond MIDI. The Handbook of Musical Codes. MIT Press. ISBN: 9780262193948.
FERKOVÁ, E. - ADAMOVÁ, S. - ŠUKOLA, M. - URBANCOVÁ, H. 2021. Computer search tool for harmonic structures and progressions in MIDI files and possible applications. Clavibus Unitis 2020, Vol. 9. No. 3. [online] https://acecs.cz/media/cu_2020_09_03_ferkova.pdf.

Keywords: Musical encoding, MIDI, harmonic analysis, MIDI extension, music, encoding

Supervisor:	doc. RNDr. Andrej Ferko, PhD.
Department:	FMFI.KAG - Department of Algebra and Geometry
Head of department:	doc. RNDr. Pavel Chalmovianský, PhD.
Assigned:	14.09.2022

Approved: 18.10.2023 doc. RNDr. Dana Pardubská, CSc.
Guarantor of Study Programme

.....
Student

.....
Supervisor

Acknowledgments: I would like to thank my supervisor Doc. Andrej Ferko for introducing me to the topic of this thesis and for guiding and inspiring me during the writing and research process. My gratitude also goes to Michal Šukola, who answered any question I had concerning his algorithmic harmonic analysis and provided me with the source code of the algorithm; and prof. Eva Ferková, who has reviewed parts of my work and helped me discover issues within it. I thank my family and friends, who have always supported me throughout my studies.

Abstrakt

Práca analyzuje rôzne digitálne kódovania hudby a porovnáva ich silné a slabé stránky. Predstavuje nové rozšírenie pre Standard MIDI File formát, ktoré umožnuje ku prehrávacím dátam MIDI súboru pridať ich harmonicku analýzu. Navyše, sme vytvorili program v jazyku Java, ktorý rozširuje .mid súbor o výsledky algoritmickej harmonickej analýzy.

Kľúčové slová: hudobné kódovanie, MIDI, MIDI rozšírenie, harmonická analýza, hudba, kódovanie

Abstract

This thesis examines numerous digital encodings of music and compares their unique strengths and weaknesses. We introduce an extension to the Standard MIDI File format which allows the coupling of the MIDI playback data with its harmonic analysis. Additionally, we have created a proof-of-concept Java program which extends a .mid file with the results of an algorithmic harmonic analysis.

Keywords: musical encoding, MIDI, MIDI extension, harmonic analysis, music, encoding

Contents

List of Figures

Introduction

Encoding music in a digital form is an arduous task. Firstly, there are many technical pitfalls to overcome: the size of the resulting file, processing efficiency or the development of the necessary software tools to facilitate the writing and reading of the format. The structure of the encoding itself needs to be general enough to accommodate diverse forms of musical expression, but simple enough to be easy to encode in and parse-through.

A musical piece has numerous aspects. To name a few: the sound, the graphical representation of the piece, performance-related information or higher analytical information, such as harmonic analysis [22]. To create a truly all-encompassing musical code would be an impossible and ultimately impractical effort. The starting point of the design process behind each musical encoding should be determining which musical aspects are to be its focus.

The primary focus of the XML-based musical encodings — MusicXML and MEI — is notation. MusicXML was designed as an interchange format between different scorewriting software [20]. As a result, its syntax is relatively simple. On the other hand, MEI has a more complex structure which can encode many more aspects of the musical piece [16].

MIDI's purpose is to facilitate music recording and playback, but is lacking in other domains. In an effort to broaden its capability, numerous MIDI extensions were developed — extensions for encoding analytical, performance or notational data beyond the scope of the original file format [22].

This thesis aims to define a new extension to the Standard MIDI File (SMF) format which would allow the coupling of the MIDI playback data with its harmonic analysis. This aim is an original approach to the problem of storing harmonic data for the subsequent statistical and musicological study of the works.

As a proof-of-concept, we have also developed two programs in the Java programming language which allow the user to add harmonic analysis data to a .mid file and subsequently read and process this data. We have used an algorithmic harmonic analysis program developed by Mgr. Michal Šukola in collaboration with prof. Eva Ferková for the analysis of the MIDI files [12].

Chapter 1

Encoding music

Attempts at encoding music are known from remote times. The earliest evidence of a musical code was unearthed in the Babylonian city of Nippur [18], dating as far back as 1400 BC. The ancient Greeks had a notation system of their own [5], as so did the ancient Hindus [15]. The standard Western notation system first appeared in its most rudimentary form in the 9th century, increasingly becoming more complex and ornate as time went on [8]. However, notations written by hand are fundamentally limited in their scope.

The introduction of the computer, with its power and memory, has provided us with great possibilities to transcribe musical pieces more completely and in far greater quantity. A total representation of music, to encode every bit of a musical composition, has been a goal pursued in the past. However, in practice, this has proven cumbersome [22]. A musical code has to make some sacrifices, prioritize certain aspects over others.

We distinguish three kinds of musical codes based on their application. Codes designed for graphical or notational purposes, codes designed for the playing of sound and codes designed for analysis [22]. In practice, there's a great difference between the notational codes and the latter two and the conversion between these formats is never as straightforward as it would appear.

This thesis is focused primarily on the MIDI file format. We devoted an entire chapter to it and its extensions. However, it is important to be acquainted with some of its many alternatives as well.

1.1 Sound-related codes

Csound is a sound and computing system, originally developed by Barry Vercoe of MIT in 1985 [1]. It's a programming language of sorts, with conditions, IF statements and variables. The user composes a musical piece much like he would write any other programming code.

A similar project currently in early development is **Dflat** [2] of Martin Ilčík, a researcher based in TU Wien. The syntax of the code is derived from C#.

Dflat code example

```
function Main() {
  @harmonies = new [] { 1, 4, 5, 3 };
  @degrees = new [] { -7, -1, 2, 4, 7 };
  @velocities = new [] { 0.75, 0.65, 0.5, 0.55 };
  @phrases = 4;
  @bars = 4;
  @velocity = 0.75;
  @degree = 0;
  @span = @phrases * @bars * 4/3;
  @Split(@phrases).Then(idx => {
    @min = idx.Index * 4;
    @max = @min + 8;
    @velocity = @velocities[idx.Index];
  });
  @Split(@bars).Then(idx => {
    @harmony += @harmonies[idx.Index];
  });
  @Split(@degrees.Length).Then(idx => {
    @degree += @degrees[idx.Index];
    @time -= idx.Index * @span + (idx.Index * 0.1 * @span);
    @span *= @degrees.Length;
  });
}
```

It would be very difficult and tedious to encode already existing longer works, such as symphonies, in this way. However, it is interesting to hear complex and pleasant-sounding compositions encoded in such compact files. Music in general follows rules and algorithms — its structure is in many ways predictable. We can take advantage of this fact when designing future digital musical encodings. These musical programming languages also offer a creative new way to compose music through the syntax and process of standard computer programming, leading to unique and interesting results.

1.2 Notation-related codes

1.2.1 DARMS

One of the earliest attempts at digitally encoding music with the goal of notation is DARMS, short for Digital-Alternate Representation of Musical Scores. Its development was begun by Stefan Bauer-Mengelberg and Melvin Ferentz in 1961 [10], the motivation being the rising cost of musical engraving. The task of encoding musical scores was soon discovered to be an arduous one, since standard musical notation is rife with exceptions and irregularities [11].

Originally developed with punch cards in mind, the DARMS encoding language has survived and developed decades after its creation, spawning many dialects with diverse application [22]. Today, DARMS sees no practical use. However, as the pioneer project in its field, which first had to wrestle with the issues of usability, irregularity and completeness, it deserves special inquiry. To learn more about DARMS and its dialects see the *Beyond MIDI book* starting at Chapter 11 [22].

1.2.2 MusiXTeX and PMX

MusiXTeX is set of music typesetting macros for TeX, based on the earlier MusicTeX [9]. It allows users to create detailed musical scores within TeX documents. Since its syntax is archaic and difficult to use, it is recommended to use the PMX preprocessor which compiles an easier PMX input language into MusiXTeX macros.

MusiXTeX example, see figure below

```
\input musixtex
\parindent10mm
\setname1{Piano}
\setstaffs12
\generalmeter{\meterfrac44}
\nobarnumbers
\startextract
\Notes\ibu0f0\qb0{cge}\tbu0\qb0g|\hl j\en
\Notes\ibu0f0\qb0{cge}\tbu0\qb0g|\ql l\sk\ql n\en
\bar
\Notes\ibu0f0\qb0{dgf}|\qlp i\en
\notes\tbu0\qb0g|\ibbl1j3\qb1j\tbl1\qb1k\en
\Notes\ibu0f0\qb0{cge}\tbu0\qb0g|\hl j\en
\endextract
\end
```

PMX example

```
2 1 4 4 4 4 0 0
1 1 20 0.12
Piano
tt
./
% Bars 1-2
c8 g+ e g c- g+ e g | d g f g c- g+ e g Rb /
c2+ e4 g | bd4- c1 d c2 /
```

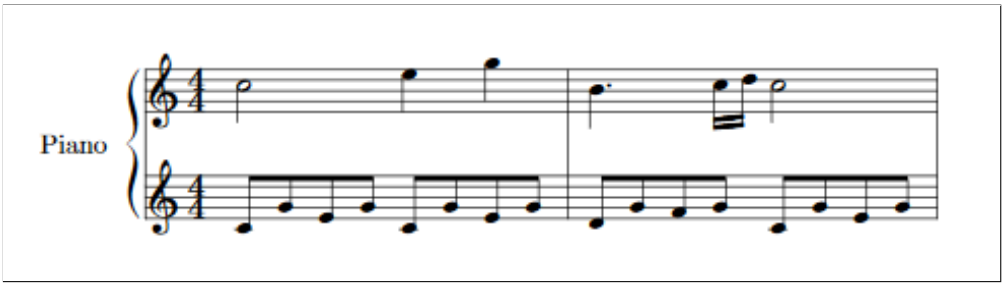

Figure 1.1: The output of the MusiXTeX and PMX encoding above. The first two bars of Sonata K545 by W.A. Mozart [19].

A PMX code consists of two parts: the preamble and the body [19]. Lines beginning with % are comments and are disregarded by the processor. The preamble contains musical and typographical specifications for the score. The first part of the preamble must contain 12 numerical parameters separated by one or more whitespaces. Their meaning are as follows:

The first two numbers specify the number of staves and the number of musical instruments respectively. The following four numbers are related to the meter. The first two represent the logical meter PMX uses to calculate the length of the bar — the first number representing the nominator and the second the denominator of the meter. The second pair merely define the appearance of the meter in the printed output. The 7th parameter specifies the number of pickup bars in the piece — a pickup bar refers to bar whose number of beats doesn't align with the ones set by the meter.

What follows is the key signature — positive integers specify the number of sharps and negative integers the number of flats. The last four numbers are related to the layout of the score: the number of pages, the number of systems, the height of a staff and the indentation of the first system from the left margin.

After the base characteristic parameters are defined, the preamble follows with a new line for every system, relating the name appearing in the indentation before the system. In the example above, the only system defined in the PMX code is named

"Piano". The next line is a string specifying the clef for each stave of the composition. The preamble ends with the path name of the directory the user wants the files written to.

The rest of the PMX file is the body [19]. It may begin with a couple of lines setting global options for the piece — this is referred to as the body's header. What follows is the code of the music input itself. The basic unit of this encoding is called a block, each consisting of 1 to 15 bars and relates to a particular staff. Staves are referenced sequentially, beginning with the lowest staff. Bars within blocks may be separated by a | symbol — this is not required, but helps with the human-readability of the code. Distinct voices within a staff are demarcated by a //, while a block ends with the / symbol.

The two main features of a PMX note are its pitch and duration. The pitch is determined by the note name and octave. The note name is encoded as a lowercase letter of the natural with the possible accidental following it — s and ss for sharp and double sharp, f and ff for flat and double flat and n for natural. The octave can be encoded in numerous ways: explicitly — a second (after the duration number) unsigned digit specifying the pitch; implicitly — inherited from the previous note; and relatively — one or more octaves higher or lower from the previous note indicated by one or more + or - signs. The basic duration without elongating dots is indicated by the first number following the note name — either 9 — double-whole note, 0 — whole note, 2 — half note, 4 — quarter note, 8 — eigth note, 1 — sixteenth note, 3 — thirty-second note, 6 — sixty-fourth note. Many more aspects of a note can be encoded, such as dots, ornaments and so on [19].

Some features of MusiXTeX are not directly accessible through the PMX programming language. However, PMX preprocessor supports the insertion of MusiXTeX code in the PMX file to alleviate this problem [19].

In conclusion, MusiXTeX and PMX is sufficient for encoding the notation of most western classical works. The PMX code is very human-readable and especially easy to write in without the need for any external editors — a consequence of its bare-bones and simple syntax. However, beyond the purpose of notation, the codes lack standardized features and structures to allow coupling the musical data with metadata for the analysis of the encoded musical works.

1.2.3 XML-based codes

Extensible Markup Language, or XML for short, is a file format and markup language designed for storing and exchanging data in a platform-independent way [24]. The base syntax of the XML file format is simple and flexible, consisting of tags, their attributes

and the data itself. In contrast to some other markup languages, such as HTML or Markdown, it does not have any predefined tags and its up to the user to introduce their own. XML schemas are descriptions of structural and data-related constraints which define an XML document type. Documents that adhere to the basic syntax of XML are called well-formed and if they also adhere to their schema, they are referred to as valid [24].

Considering the qualities of the format, it is not hard to imagine XML would lend itself well as the groundwork for notation-focused encoding of music. We provide an overview of two such codes — the MusicXML format and MEI.

MusicXML

MusicXML is a XML-based format for encoding musical scores. Developed by the W3C Music Notation Community Group, its purpose is to facilitate the interchange of musical information among different music-related software [20], thought it may also be used as a standalone sheet music distribution format [13]. Its first version was released in January 2004. Today, it's supported by 271 different programs, including scorewriting software, like Musescore and Sibelius, as well as digital audio workstations (DAWs), such as Cubase and Logic Pro [23]. Below is a simple example of a MusicXML document:

MusicXML 4.0 example [13]

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE score-partwise PUBLIC
    "-//Recordare//DTD MusicXML 4.0 Partwise//EN"
    "http://www.musicxml.org/dtds/partwise.dtd">
<score-partwise version="4.0">
  <part-list>
    <score-part id="P1">
      <part-name>Music</part-name>
    </score-part>
  </part-list>
  <part id="P1">
    <measure number="1">
      <attributes>
        <divisions>1</divisions>
        <key>
          <fifths>0</fifths>
        </key>
        <time>
```

```
            <beats>4</beats>
            <beat-type>4</beat-type>
          </time>
          <clef>
            <sign>G</sign>
            <line>2</line>
          </clef>
        </attributes>
        <note>
          <pitch>
            <step>C</step>
            <octave>4</octave>
          </pitch>
          <duration>4</duration>
          <type>whole</type>
        </note>
      </measure>
    </part>
  </score-partwise>
```

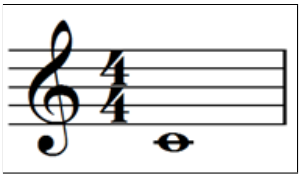

Figure 1.2: A representation of the MusicXML coding above in standard notation

The root element type of this particular MusicXML document is <score-partwise>. This means the document is divided into musical parts with each part being further divided by measures. This is in contrast with the <score-timewise> element, which divides the document into measures made up of parts. Every MusicXML document begins with the <part-list> header which lists off all the parts of the score and assigns them a unique ID.

Since this a <score-partwise> document, what follows are the parts themselves made up of their measures. The <attributes> element of a measure contains information required to interpret its musical content, such as the key, time signature, clef and so on. This particular measure includes one other element — <note>. The pitch of the note is encoded in Scientific pitch notation (SPN), specifying the note name and the octave separately. The division value is 1 per quarter note, hence the the duration value of a whole note is 4. Besides pitch and duration information, we can also express

many other qualities of a note, including the beam, note-head or playback information.

The document may contain many other elements. Of interest to us are the ones related to harmonic analysis. The <harmony> element assigns a chord to a musical section [13]. The chord is identified by the kind of chord, e.g. major, half-diminished, etc., and its root note. The root note is to be encoded in two ways, either by the root note name or the its position within the scale. Borrowed chords may be encode as a sequence of chords, each with their own <kind> element. For an example, a V to II chord would be represented by chord with a 5 numeral and second one with a 2 numeral.

MusicXML <harmony> element example

```
<harmony>
    <root>
        <root-step>C</root-step>
    </root>
    <kind use-symbols="yes">major-seventh</kind>
</harmony>
```

The <grouping> element allows for diverse forms of analysis [13]. Its parent attribute is a measure in the partwise format, or a part in the timewise format. Its only, but syntactically required attribute is @type, specifying the scope of the element — the possible values are: start, stop and single.

The <grouping> element may contain one or more <feature> elements, which hold the analysis information itself. The @type attribute may be used to differentiate between the kinds of analysis. Apart from what has already been described, MusicXML currently defines no further specification on the structure of the values contained in the <grouping> element — though the authors do mention the possibility of such specification in future MusicXML versions [13].

MusicXML <grouping> element example

```
<grouping type="start">
        <feature type="motif">
                Fate knocking at the door
        </feature>
</grouping>
    ...
<grouping type="stop"/>
```

A regular MusicXML file can be quite large — much larger than its MIDI or application-specific file counterparts. To alleviate this issue a zip-based compressed

MusicXML file format was introduced, reducing the standard MusicXML file roughly to the size of a corresponding MIDI file [13].

MEI

The Musical Encoding Initiative (MEI) is an organization leading a community-driven effort to define an open-source, versatile and platform-independent musical encoding with a machine-readable structure [16]. The term MEI also describes the XML-based file format developed by the organization. The following MEI example relates the same information as the previous MusicXML example:

MEI 5 example

```
<mei xmlns="http://www.music-encoding.org/ns/mei">
<meiHead>
  <fileDesc>
    <titleStmt>
      <title/>
    </titleStmt>
    <pubStmt/>
  </fileDesc>
</meiHead>
  <music>
    <body>
      <mdiv>
        <score>
          <scoreDef meter.count = "4" meter.unit ="4">
            <staffGrp>
<staffDef clef.shape="G" clef.line="2" n="1" lines="5"/>
            </staffGrp>
          </scoreDef>
            <section>
              <measure>
                <staff n="1">
                  <layer>
                    <note pname="c" oct="4" dur="1"/>
                  </layer>
                </staff>
              </measure>
            </section>
        </score>
```

```
        </mdiv>
      </body>
    </music>
  </mei>
```

The code example above uses <mei> as its root element. This element must begin with the <meiHead> header, which may include meta-data related to the encoding, such as its title, the author of the encoding, the composer of the music, the publisher and so on. It's followed by the <music> element, containing the musical text itself. The musical text can describe anything from a simple piano prelude to complex forms of music, such as operas, or entire anthologies of a composer's life work [17].

The <score> elements contains an entire score. It begins with the <scoreDef> element, which holds important meta-data for the score. In this particular example, the meter, staff and clef is encoded.

MEI provides elements and attributes for the systematic encoding of different forms of musical analysis. It offers many attributes to describe relationships between musical elements. For example the @sameas attribute, denoting the sameness of musical elements, or the @next and @previous attributes, which allow the user to define a sequence of musical elements [17]. The @inth attribute may be used to encode harmonic intervals between pitches occurring at the same time.

MEI separates the definition of harmonic labels and their assignment to musical sections. The <chordTable> element holds the chord definitions. It may contain multiple <chordDef> elements, which describe the chords themselves. Within the <chordDef> element, each pitch belonging to the chord is represented by a <chordMember> element. The <chordDef> element should include an @xml:id attribute, assigning the chord definition a unique ID with which it is to be referenced in following <harm> element declarations within the score using the @chordref attribute [17].

<chordDef> element example

```
<!-- Chord defined in scoreDef -->
<chordDef xml:id="harmonychordA">
<chordMember oct="2" pname="a"/>
<chordMember oct="3" pname="e"/>
<chordMember accid.ges="s" oct="4" pname="c"/>
<chordMember oct="4" pname="e"/>
<chordMember oct="4" pname="a"/>
</chordDef>
<!-- Later in musical text -->
<harm chordref="#harmonychordA" tstamp="1">A</harm>
```

The MEI encoding distinguishes itself from the more widespread MusicXML by its expressive power. MusicXML's primary goal is to be an interchange format between different notational software. MEI can also serve this purpose, but beyond this can encode very diverse and elaborate notational information coupled with its intellectual content and interpretation in a structured and systematic manner. As an example of its expressivity, the MEI format supports notation systems beyond the Common Western Notation, such as medieval and renaissance-era notations [16].

This is possible by a complex, abstractive and versatile schema definition which grants the user great freedom in what they are able to encode. As a result, the MEI encoding is well suited for many academic purposes or as a standalone musical encoding format for digital libraries and repositories.

1.3 Summary

Each of the musical encodings mentioned have their unique strengths and weaknesses stemming from their intended use case. The musical programming languages, like Csound and Dflat, introduce the option of encoding music through the language of algorithms. This approach can be used to significantly reduce the size of the files and opens up the possibility to place harmonic concepts, such as chord progressions, at the lowest level of the musical encoding — to build the musical piece with all its particularities around a set harmonic structure rather than the inverse of assembling the harmonic concepts post hoc from an unstructured stream of notes. However, it's difficult to encode already existing works in this manner.

The PMX preprocessor for the MusiXTeX encoding offers simple and very human-readable notational code, but lacks other features. The two XML-based codes — MusicXML and MEI — are excellent notation-focused musical encodings. The MEI encoding in particular offers many features for all kinds of complex musicological analysis. However, the formats are too verbose for cases when notational data is not important.

Chapter 2

MIDI

MIDI is an acronym for Musical Instrument Digital Interface. It's a technical standard that facilitates transmission, recording and storage of musical data [22].

It was originally developed as a real-time protocol to enable communication between separate hardware (e.g. an electronic keyboard connected to a computer) in the 1980s [6]. Before the widespread adoption of MIDI, each manufacturer had their own proprietary standard to synchronize instruments, which proved cumbersome and limiting to the growth of the electronic music industry.

The Standard MIDI File (SMF), which it describes, holds no audio information. Instead it contains a sequence of MIDI (and non-MIDI) events such as note on, note off and so on. It is the responsibility of the software which plays the MIDI file to translate these events into sound. [6].

The SMF file contains 2 kinds of sections or chunks [22].

2.1 The Standard MIDI File Format

2.1.1 The Header chunk

A SMF file begins with a 14 byte long Header chunk. It includes information that pertains to the whole composition. [22] The syntax of the Header chunk is as follows [14]:

Header chunk syntax

```
Header Chunk := "MThd" + <Header Length> + <Format> +
                + <Number of Tracks> + <Time Division>
```

The "MThd" string (4 bytes) specifies that this is a Header chunk. Header length (4 bytes) is always 6, referring to the 6 bytes that follow. The format (2 bytes) can be one of three values: 0 - the MIDI file has exactly one track; 1 - the MIDI file contains multiple tracks forming one sequence; 2 - the file has multiple tracks but each one

represents an independent sequence e.g. with their own tempo. The number following the format value refers to the number of tracks contained in the file (2 bytes). The time division (2 bytes) specifies how to convert MIDI ticks to real time: either "ticks per beat" or "frames per second".

The Header file is followed by one or more Track chunks.

2.1.2 The Track chunk

A Track chunk contains the data of a single MIDI Track. Below is its syntax [14]:

Track chunk syntax

```
Track Chunk := "MTrk" + <Track Length> + <MTrk Event> + ... +
             + <MTrk Event> + <End of Track Meta-event>
```

The "MTrk" string (4 bytes) specifies this is a Track chunk. Track length (4 bytes) describes the length of the Track in question. What follows is a sequence of Track chunk events, always ending with the End-of-track Meta-event. The syntax of a MTrk event is as follows [14]:

MTrk event structure

```
MTrk Event := <Delta Time> +
            + <MIDI Event> OR <SysEx Event> OR <Meta-event>
```

Delta time refers to the time passed after the previous event in units specified in the Header chunk. Its byte-length can vary. There are three types of events: MIDI events, Meta-events and System-exclusive events.

A) MIDI Events

A MIDI event is any Channel MIDI message. It begins with a status byte — the first 4 bits specify event type, the last 4 the channel (instrument output). The status byte may be omitted if the previous event was a MIDI channel message with the same status, this is known as the running status.

Two MIDI events are of particular importance to us — the Note On and Note Off events. Their syntax is outlined below:

Note On and Note Off event syntax

```
Note On Event  := 1001nnnn 0kkkkkkk 0vvvvvvv
Note Off Event := 1000nnnn 0kkkkkkk 0vvvvvvv
```

The first byte of the event is the status byte. What follows are 2 data bytes: the first specifying the key of the note, the second the velocity with which its to be played.

B) Meta-events

Meta-events are used to relate information such as key signature, tempo, instrument name, copyright notice etc. [22].

Meta-events are distinguished from other events by their status byte, which has a value of 0xFF. This is followed by a type byte, variable length value byte and the data of the message itself [21].

Meta-event syntax

```
Meta-event := 0xFF + <Meta-event Type> +
              + <Length Byte> + <Message Bytes>
```

Currently, the standard MIDI file format defines 15 different Meta-event types:

Type	Event	Type	Event
0x00	Sequence number	0x20	MIDI channel prefix assignment
0x01	Text event	0x2F	End of track
0x02	Copyright notice	0x51	Tempo setting
0x03	Sequence or track name	0x54	SMPTE offset
0x04	Instrument name	0x58	Time signature
0x05	Lyric text	0x59	Key signature
0x06	Marker text	0x7F	Sequencer specific event
0x07	Cue point		

Figure 2.1: Meta-event types [21]

Two Meta-events express information that is especially important for the purpose of harmonic analysis:

First, the Time signature Meta-event which holds information of the current time signature of the composition. Its syntax is as follows [14]:

Time signature Meta-event syntax

```
TimeSIG := 0xFF 0x58 0x04 0xNN 0xDD 0xYY 0xZZ
```

The first three bytes specify that this is a Meta-event of the Time signature type with 4 data bytes. The following two bytes represent the nominator and denominator of the time signature. The denominator is encoded as a negative power of two. The 0xYY parameter represents the number of MIDI ticks in a metronome tick, while 0xZZ specifies the number of notated 32nd-notes in a MIDI quarter-note.

The second Meta-event is the Key signature:

Key signature Meta-event syntax

```
KeySIG := 0xFF 0x59 0x02 0xYY 0xZZ
```

Here the 0xYY byte expresses the number of flats (when the number is negative) or sharps (when the number is positive) in the specified key. The last byte represents if the key is major or minor, 0 or 1 respectively. For examples if 0xYY was to hold the value -1 and 0xZZ a 0, the specified key would be F major.

MIDI processing programs should be able to ignore Meta-events which are not defined in the table above, as more Meta-events may be introduced in the future [14]. Syntax requires Meta-events to be stored within individual Track chunks, even if they describe events pertaining to all tracks of the MIDI sequence.

C) System-exclusive Events

System-exclusive or SysEx events pertain to a particular hardware system and manufacturer of the receiving MIDI device. Their lengths are variable and can be quite extensive [22].

They begin with the status byte of 0xF0 followed by a manufacturer ID byte assigned to them by the International MIDI Association. If a developer does not have an assigned ID, they can either use someone else's — with permission from the manufacturer and on the condition that they respect the format of the SysEx messages - or use the ID 0x7D, which is reserved for non-commercial use and has no defined message formats [4].

2.2 SMF Extensions

The standard MIDI file format, designed primarily for the recording and playing of music, lacks many features necessary for notation or harmonic analysis [22]. For example, the MIDI doesn't differentiate between enharmonic notes (notes of the same pitch, but with different names e.g. Fb and E). To alleviate these issues, many MIDI extensions were defined, adding to or modifying events of the original MIDI standard.

These extensions differ in their purpose and degree of compatibility with the standard MIDI file format. We will examine a few existing ones which are described in the Beyond MIDI book [22] and, in the chapter four, introduce a new extension designed for the purpose of harmonic analysis.

2.2.1 NoTAMIDI

The NoTAMIDI file extension was developed at Oslo University in the late 1980s [22]. It was presented to the International MIDI Association and the MIDI's Manufacturers' Association, but neither took action. The specifications are now in public domain.

The extension adds 7 new Meta-events and modifies 2 existing ones. The figure below shows all Meta-event defined in the extension.

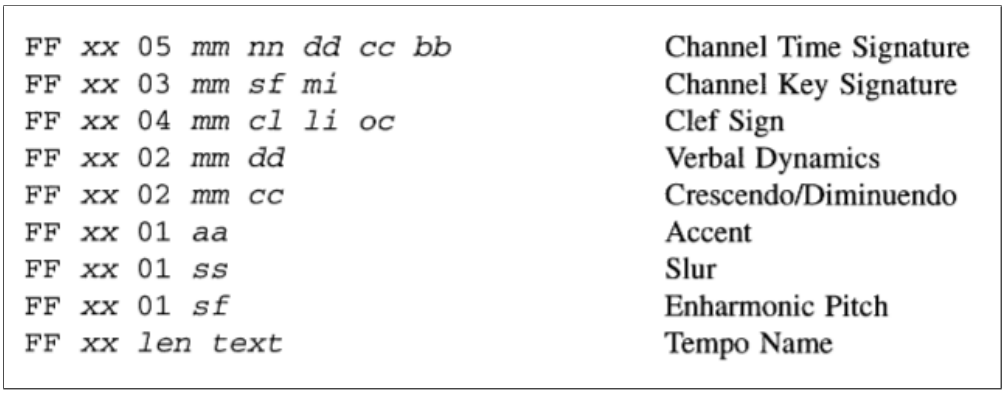

Figure 2.2: NoTAMIDI Meta-events [22]

A notable feature of the NoTAMIDI extension is the introduction of a track (channel) byte for Meta-events [22]. The extension modifies the original Meta-events of Time signature and Key signature to specify the track they are assigned to. The extension also defines ways to add notation-related information such as clef signs, accents, dynamics and enharmonic pitch to the MIDI file. The value of type bytes of the new Meta-events are not defined.

The enharmonic pitch is expressed by the Enharmonic Pitch Meta-event which may precede a note [22]. It doesn't change the pitch of the note, rather expresses how it is to be notated. The Meta-event contains one data byte. The proposed codes are: 0 — sharp sign, 1 — flat sign, 2 — double sharp sign, 3 — double flat sign and 4 — natural. The current key signature should be taken into account when adding Meta-events of this type, e.g. a F sharp note does not require a sharp sign in the G Major. Pairs of notes and Meta-events which are impossible, such as a D note expressed with one sharp in C Major, are to be ignored.

2.2.2 MIDIPlus

MIDIPlus is a proposed MIDI extension designed for the purpose of musical notation [22]. It encodes extra-musical information within the velocity bytes of Note On and Note Off events.

The enharmonic spelling of notes is to be stored in the last two low-order bits of the Note On event. Every MIDI note can be spelled 3 different ways using 2 or fewer accidentals. Figure 2.3 shows the proposed values of the spellings.

Spellings with more than two accidentals can not be encoded in this way, however they are exceptionally rare in musical notation. The resulting MIDI files will still sound relatively unchanged as the differences in velocities are negligible for the listener.

Value	Key Number/Pitch Value											
	60	61	62	63	64	65	66	67	68	69	70	71
0	pitch spelling undetermined											
1	D𝄫	D♭	E𝄫	F𝄫	F♭	G𝄫	G♭	A𝄫	A♭	B𝄫	C𝄫	C♭
2	C	C♯	D	E♭	E	F	F♯	G	G♯	A	B♭	B
3	B♯	B♯♯	C♯♯	D♯	D♯♯	E♯	E♯♯	F♯♯	F♯♯♯	G♯♯	A♯	A♯♯

Figure 2.3: MIDIPlus Enharmonic pitch values [22]

Similarly, the extension also proposes slurs to be encoded using 4 bits of a velocity byte through "Slur On" and "Slur Off" flags, though it can not encode more than 3 concurrent slurs.

The authors of the extension noted that the Note Off event velocity offers lots of room for encoding information, like stem direction, without encroaching on the standard uses of MIDI files as most MIDI players ignore the value of this byte altogether.

2.2.3 Expressive MIDI

The extension introduces a new System-exclusive event. Its ID is set at 0x7D (the reserved number for non-commercial uses) and allows the user to store multitude of notation-related data. The *expEvent* is placed immediately after the Note Off event of the related note. Symbols which are not related to notes, such as bar lines or clefs, can be placed freely [22].

The syntax of an expEvent is:

expEvent syntax

```
expEvent := 0xF0 0x7D 0XXYYYYY 0ZZZZZZZ 0xF7
```

The two data-bytes encode one symbol. If multiple symbols were to relate to one note, the expEvent may contain multiple data-bytes. The variable XX is the encoding-table number. YYYYY specifies symbol-specific parameters. The last data-byte encodes the symbol number within the encoding-table.

The extension defines three encoding-tables. The first two contain isolated symbols, meaning their shapes are invariable. The third contains extensible symbols, such as slurs or pedal markings [22].

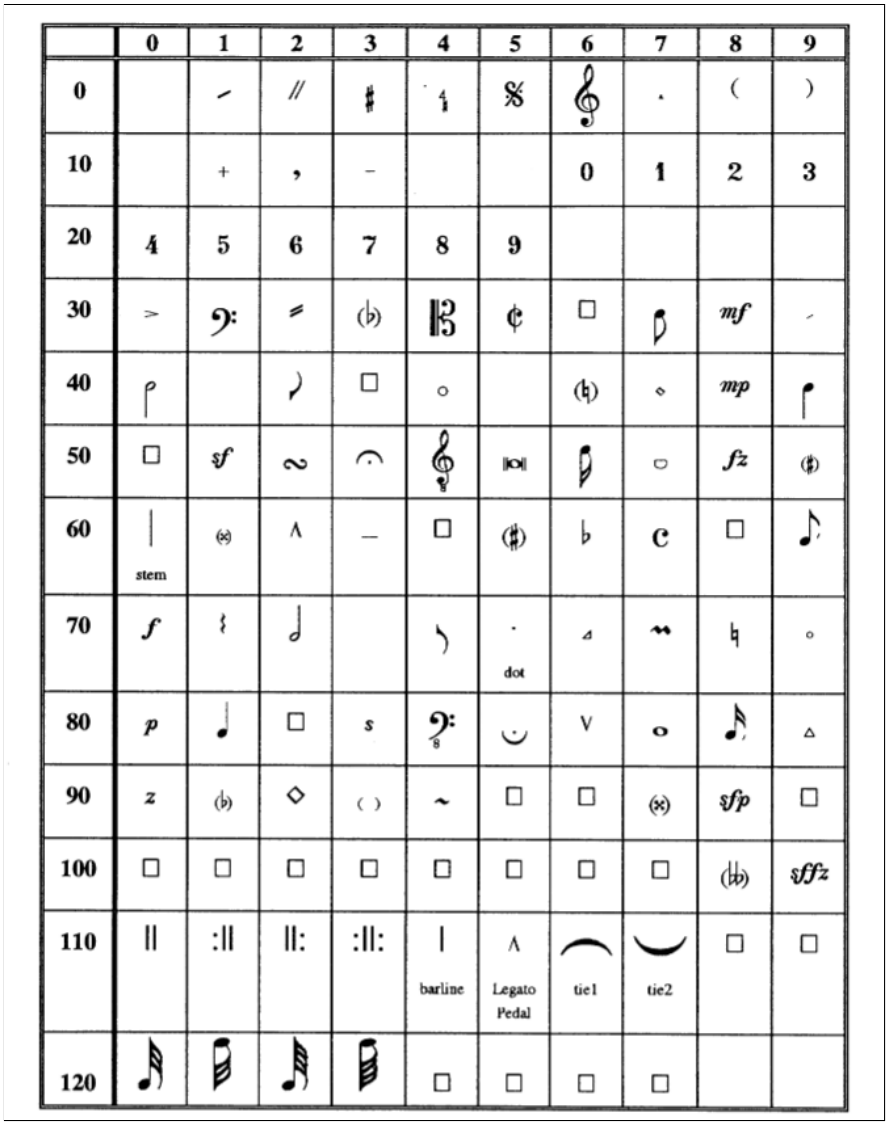

Figure 2.4: Expressive MIDI encoding-table 0 [22]

	0	1
Y_1	Above a symbol	Below a symbol
Y_2	On a symbol	After a symbol
Y_3	Isolated	Symbol attachment
Y_4	Normal	Miniature (grace)
Y_5	Normal	Crushed

Figure 2.5: Expressive MIDI bit-flags for isolated symbols [22]

2.3 Summary

Although the Standard MIDI File format was created for the purpose of sound recording and playback, it can serve as a stable foundation for other purposes, such as musical notation or analysis. The format is quite versatile and offers many ways to extend it beyond its original use case.

Despite this, we struggled to find MIDI extensions other than the four described in the Beyond Midi book. Of these four MIDI extensions[1], none were designed for the purpose of harmonic analysis. At best they offered a few features useful for analysis, such as enharmonic spelling, but all lacked ways to encode information crucial in the domain of harmonic analysis, such as chords.

[1]The one MIDI extension present in the Beyond MIDI book but not described in this thesis is Augmented MIDI, designed to include performance-related information [22].

Chapter 3

Harmonic analysis

3.1 Harmony

The main characteristic of tonal music is the alternation of an increasing tension and its subsequent resolution [12]. Though there are many aspects one can observe in a musical composition, such as melody, tempo or timbre, this ebbing movement is the domain of harmony.

3.1.1 Chords and their Harmonic function

Harmony is the simultaneous sounding of different tones. Three or more tones playing together is known as a chord, though we also recognize broken chords as chords in most melodic contexts. Chords are the fundamental elements of inquiry in harmonic analysis, the bearers of tension and resolution [12], and in turn the most important elements in the expressive and emotional force of a musical piece. Harmonic analysis focuses on three chords relative to the key of the musical section. These are the Tonic, the Subdominant and the Dominant [12]:

- The **Tonic**'s root note is the beginning of the scale. Its function is stability, resolution, the feeling of home. It serves as the centre of the piece. Musical sections usually start and end in the Tonic.

- The **Subdominant** chord begins at the fourth note of the scale. Its function is centrifugal, stepping away from the stability of the Tonic. It is usually, but not always, followed by the Dominant.

- The **Dominant** provides the most amount of tension in the piece. It begins in the fifth note of the scale. Its function is in turn centripetal: the listener feels a subjective desire for its resolution in the Tonic.

There are other chords which can be assembled from the notes of the scale. In harmonic analysis, these chords are said to serve one of the three fundamental chord functions, depending on the notes they are made of and the musical context they are surrounded by [12]. For example, the chord beginning in the 6th degree of the scale can alternately serve the function of the Tonic, if preceded by the Dominant; or the Subdominant, if it's to resolve in to the Dominant. Similarly the chord of second degree can substitute for the Subdominant [12].

A harmonic cadence is the progression of two or more chords that concludes a musical section [7]. It may be compared to rhyme at the end of a line in a work of poetry. The frequency of certain cadence cycles combined with the uses of particular tonal keys and chord class structures may be statistically significant in the differentiation between musical styles and genres [12].

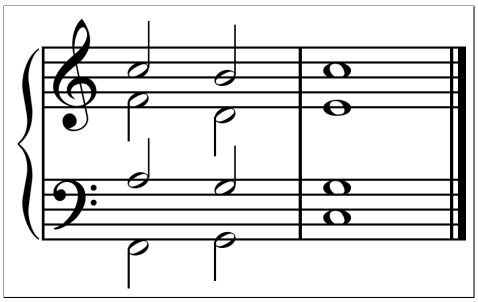

Figure 3.1: A IV-V-I chord progression in C

3.2 Algorithmic Harmonic analysis

To analyze large repositories of musical works by hand is implausible. Putting aside the great amount of time and effort it would take, the results of such harmonic analysis would be prone to mistakes and subjectivity. For this reason, many harmonic analysis algorithms were developed.

The algorithm we will use was created by Mgr. Michal Šukola in collaboration with prof. Eva Ferková [12]. The algorithm takes in a MIDI file as its input. It detects chord classes, keys and harmonic functions of discrete musical sections. The result is a Harmonic analysis of the composition viewable in the Sibelius program. Implemented in Java, it works in three phases [12].

3.2.1 The Algorithm

First phase: Chord-class analysis

In the first phase, the algorithm transforms the MIDI file format, to allow it to work with higher structures such as measures, measure sections, beats, key signatures and

time signatures. Each measure is divided in to smaller sections for better chord detection.

Each measure section is read and searched for possible occurrences of chords, including chords in arpeggiated form (a chord played as a series of ascending or descending notes). They are compared against each other, taking into consideration probability and the weight of chord tones. Melodic tones are eliminated through the examination of prevailing length of tones in a measure and whether they are not at a distance of a third from other tones.

Second phase: Key analysis

This phase uses the results collected by the first phase with additional information from the score, such as key signatures.

For a known key signature from the beginning of a processed section, the major and minor key pair is found and either confirmed or rejected, depending on the corresponding major or minor chord. It reads tones and detected chords and determines key depending on the current state. If no key is detected, the algorithm skips to the next section succeeding the already processed part of the score.

Some chord classes identify the key/scale directly. If the key is known during score processing, the algorithm assigns the same key up to the moment a tone outside of the key is found in the measure section. It then seeks its confirmation and applies the same process for chords that determine the key.

If it is unable to find a key after this processes, it tries to detect the scale structure by examining the closest group of seven tones to see if they form a major or minor scale.

Third phase: Harmonic functions identification

The third phase uses the data from the previous two phases. The harmonic functions are determined according to the scale degree of the root key belonging to the detected chord.

3.2.2 Algorithm output

The results of the algorithm can be saved in the form of a CSV text file. Here is the output file describing the first five measures of the Turkish March by W.A. Mozart:

Turkish March harmonic analysis

```
1:4.0:  null:null:A MOLL:null:null:m.mid
2:4.0:  null:null:A MOLL:null:null:m.mid
```

```
3:1.75: A MINOR_MAJOR_SEVENTH:TERZQUART(3,4):A MOLL:T (I):[T(I), II
        , D (V),T (I)]:m.mid
3:3.5: B DIMINISHED_TRIAD:ROOT(5):A MOLL:II:[T (I), II, D (V–), T (
        I)]:m.mid
3:4.0: null:null:A MOLL:null:[T (I), II, D (V), T (I)]:m.mid
4:0.75: E MAJOR_TRIAD:ROOT(5):A MOLL:D (V):[T (I), II, D (V), T (I)
        ]:m.mid
4:3.75: A MINOR_MAJOR_SEVENTH:TERZQUART(3,4):A MOLL:T (I):[T (I),
        II, D (V), T (I)]:m.mid
4:4.0: E MINOR_TRIAD:ROOT(5):A MOLL:D (V):null:m.mid
5:1.5: null:null:A MOLL:null:null:m.mid
5:1.75: E MINOR_TRIAD:ROOT(5):A MOLL:D (V):null:m.mid
5:3.25: null:null:A MOLL:null:null:m.mid
5:3.5: E MINOR_TRIAD:ROOT(5):A MOLL:D (V):null:m.mid
5:4.0: null:null:A MOLL:null:null:m.mid
```

Every line corresponds to a distinct harmonic section. The first column specifies the measure. The second expresses the end of the section within the measure in quarter notes. The third and fourth column specify the chord class and the inversion of the chord respectively. The fifth column assigns a key to the section, while the sixth column specifies the section's harmonic function. The next column relates the particular cadence the section is a part of. Finally, the last column references the name of the analyzed MIDI file.

Chapter 4

HarmonicMIDI extension

In this chapter we introduce a new MIDI extension facilitating the storage of harmonic analysis data. Of the current available options for musical encoding, we believe the SMF is the best choice for this purpose.

Firstly, the SMF is an industry standard. Nearly all DAWs and scorewriting programs offer the option of exporting the musical compositions to a .mid file. There are many large repositories of classical (and non-classical) works available on the internet, such as kunstderfuge.com [3]. In turn, many software tools and libraries were developed for manipulating MIDI files.

Secondly, the syntax of the format itself has many advantages, being compact, easily parsable and extendable. The Note On and Off events combined with their timing information are ideal atomic elements on which harmonic analysis can be performed and higher musicological concepts can be defined.

4.1 HarmonicMIDI definition

The HarmonicMIDI is a SMF extension designed for the purpose of harmonic analysis. Three technical aims influenced our design process: firstly, to ensure total compatibility with SMF; secondly, to not in any way hinder the playback of the files; finally, for the extension to be easily applicable and removable for any existing SMF file.

Our extension defines new Meta-Events. Apart from their versatility, the advantage of using Meta-Events is that, unlike SysEx events, they are not sent over ports to the MIDI players. As in the SMF, our Meta-Events begin with the 0xFF status-byte, a type-byte and a length-byte. Only 15 type-byte values are defined in the SMF, though more may be introduced in the future.

All our Meta-Events begin with the same type-byte specified in the Extension Tag Meta-Event. To differentiate in-between our Meta-Events, the first data-byte is used as a second HarmonicMIDI type-byte. There are thus 256 Meta-Events, which can

be defined in this way. This format also leaves space to include multiple different extensions in the MIDI file, all beginning with a different unreserved type-byte. Unless specified otherwise, all HarmonicMIDI Meta-Events are to be placed in the first track of the MIDI file.

4.1.1 Extension Tag Meta-Event

We encourage this Meta-Event and the aforementioned format to be used in future Meta-Event-based MIDI extensions. It specifies the particular Meta-Event extension included in the MIDI file, as well as its designated type-byte. The Meta-Event is to be placed at time 0 in the MIDI file, beginning with an unreserved type-byte. The format of the Meta-Event is as follows:

Extension tag Meta-Event syntax

0xFF + <Type−byte> + <Length−byte> + 0x00 + ASCII STRING BYTES

The ASCII String-bytes convey the name of the MIDI extension in question. In the case of the HarmonicMIDI extension, the code would be "hmidi". All subsequent Meta-Events of the extension are to begin with the same type-byte as the initial Extension Tag Meta-Event.

4.1.2 Enharmonic Pitch Meta-Event

The Enharmonic Pitch Meta-Event expresses the enharmonic spelling of a particular note. It applies to the first Note On event following its declaration. Its syntax is:

Enharmonic pitch Meta-Event syntax

0xFF + <Type−byte> + <Length−byte> + 0x03 + <Accidental−byte>

Its HarmonicMIDI type-byte value is 0x03. What follows is one accidental-byte, which specifies the kind and the amount of accidentals the note is to be written with. Negative numbers represent the number of flats, positive numbers — sharps and zero relates that the note is spelled with no accidentals. Combinations of notes and accidentals which are impossible are to be ignored. Not all Note On events require an Enharmonic Pitch Meta-Event. Most of the time the spelling of the note is straightforward and relates to the current key signature.

4.1.3 Chord Section Meta-Event

This Meta-Event declares a beginning of a chord section. The syntax of a non-empty Chord Section Meta-Event is:

Chord section Meta-Event

```
0xFF + <Type-byte> + <Length-byte> + 0x01
     + <Note-byte> + <Chord Type-byte> +
     + (<Inversion-byte>)
```

Its special type-byte value is 0x01. The next byte encodes the note of the chord — the first 4 bits the accidental: 0x0 for none, 0x1 for one sharp and 0x2 for a flat; the last 4 bits the natural: 0x0 for C, 0x1 for D and so on. The next byte specifies the chord type. The HarmonicMIDI extension defines 17 different chord types. Their names and associated values are written below:

0x00	MAJOR TRIAD
0x01	MINOR TRIAD
0x02	AUGMENTED TRIAD
0x03	DIMINISHED TRIAD
0x04	DOMINANT SEVENTH
0x05	DIMINISHED SEVENTH
0x06	DIMINISHED MINOR SEVENTH
0x07	MAJOR SEVENTH
0x08	MINOR SEVENTH
0x09	AUGMENTED SEVENTH
0x0A	MINOR MAJOR SEVENTH
0x0B	DOMINANT SEVENTH INCOMPLETE
0x0C	DOMINANT SEVENTH ALT INCOMPLETE
0x0D	MAJOR SEVENTH INCOMPLETE
0x0E	DIMINISHED SEVENTH INCOMPLETE
0x0F	DIMINISHED MINOR SEVENTH INCOMPLETE
0x0G	MINOR MAJOR SEVENTH INCOMPLETE

The user can also define their own chord types using the Chord Type Declaration Meta-Event. We will describe its syntax in the next subsection. The last byte is optional and specifies the inversion of the chord, e.g. it holds the value 0x00 for the root position, 0x01 for the first inversion, 0x02 for the second and so on.

This Meta-Event is to be place at the time of the chord section's beginning. The next Chord Section Meta-Event declares the end of the previous section and the start of a new one. The Chord Section also ends with the end of the track. An empty Chord Section Meta-Event is used to declare an unassigned chord section. The syntax of the message is similar to the one above, except it only contains one data-byte — the HarmonicMIDI type-byte with the value of 0x01.

4.1.4 Chord Type Declaration Meta-Event

This Meta-Event is used to define a chord type not present in the 17 basic chord types defined in the HarmonicMIDI extension. Its syntax is:

Chord Type Declaration Meta-Event

```
0xFF + <Type-byte> + <Length-byte> + 0x54 + <ID-byte> +
     + <Postion-byte> + <Postion-byte> + ... +
     + <Postion-byte> + 0xFF + (ASCII STRING BYTES)
```

The ID byte defines the value the chord type is to be referenced with in the Chord Section Meta-Event. Values 0-16 are already occupied by the basic 17 chord types defined in the extension. What follows are multiple position bytes which specify the number of half-steps between the notes of the chord in its root position. This sequence is terminated by a 0xFF byte. Optionally, a number of bytes may follow, allowing the user to assign a name to the chord.

For example, the Chord Type Declaration Meta-event defining the "Ode-to-Napoleon" hexachord would be:

"Ode-to-Napoleon" hexachord definition

```
0xFF + <Type-byte> + 0x05 + 0x54 + <ID-byte> + 0x01 + 0x02
     + 0x01 + 0x03 + 0x01 + 0xFF
```

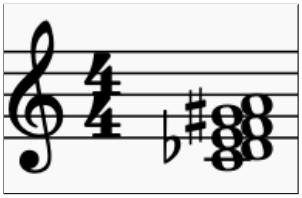

Figure 4.1: The "Ode-to-Napoleon" hexachord set at C

The Chord Type Declaration Meta-Event is to be placed at the time zero in the first track of the MIDI file, before any Chord Section Meta-Event instances.

4.1.5 Harmonic Function Meta-Event

This is an optional and complementary Meta-Event to the Chord Section Meta-Event. It is to be placed at the same time as the beginning of the chord section and its scope ends with the end of the section. Its syntax is:

Harmonic Function Meta-Event

```
0xFF + <Type-byte> + <Length-byte> + 0x02
     + <Harmonic Function-byte>
```

The HarmonicMIDI type-byte is set at 0x02. The next data-byte specifies its position within the current key, beginning with 0, and, in turn, its harmonic function, e.g. 0x00 for the tonic (I), 0x03 for the subdominant (IV) and 0x04 for the dominant (V).

4.1.6 HMIDI Key Signature Meta-Event

This Meta-Event declares the same information as the original Key Signature Meta-Event. The key itself is encoded in the same way — with one byte signifying the amount and type of the accidental and the other — if the scale is major or minor. The syntax of a non-empty HMIDI Key Signature Meta-Event is as follows:

HMIDI Key Signature Meta-Event

```
0xFF + <Type-byte> + <Length-Byte> + 0x04 +
     + <Accidnetal-Byte> + <Major/Minor-byte>
```

A non-empty HMIDI Key Signature Meta-Event differs from the the standard in its type-byte and an extra data-byte relating its HarmonicMIDI type-byte, which is set at 0x04. As in the Chord Section Meta-Event, the scope of the Meta-Event ends at with the declaration of a new HMIDI Key-Signature. An empty HMIDI Key Signature, relating an unassigned key signature section, contains the HarmonicMIDI type-byte as its only data-byte.

Apart from allowing the definition of an unassigned key, the rationale behind using the HMIDI Key Signature rather than the standard Key Meta-Event defined in the SMF is to not change the original key signature definitions of a MIDI file in order to keep the Meta-Events of the harmonic analysis distinct.

4.1.7 Scale Declaration Meta-Event

Though beyond the scope of functional harmonic analysis, which only deals with the major and minor scales, we thought it to be useful to be able to declare other scales. We introduce the Scale Declaration and the Scale-Key Meta-Events for this purpose.

The Scale Declaration Meta-Event defines a scale and assigns it a unique ID to be used by the Scale-Key Meta-Event. The syntax of the Scale Declaration Meta-event is as follows:

Scale Declaration Meta-Event

```
0xFF + <Type-byte> + <Length-byte> + 0x64 + <ID-byte> +
     + <Step-byte>  + <Step-byte> + ... + <Step-byte> +
     0xFF + (ASCII STRING BYTES)
```

The HarmonicMIDI type-byte is set at 0x64. The next byte is the ID-byte. What follows are a sequence of up to 7 step-bytes specifying the number of half steps between

the notes of the scale, ending with a 0xFF byte. The user may also assign a name to the declared scale. For example, the Scale Declaration Meta-Event declaring the modern Phrygian mode would be:

Phrygian Scale Declaration Meta-Event

```
0xFF + <Type-byte> + 0x07 + 0x64 + <ID-byte> + 0x01 + 0x02 +
     + 0x02 + 0x02 + 0x01 + 0x02 + 0x02 + 0xFF
```

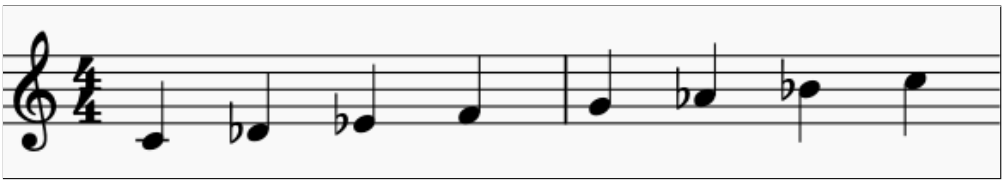

Figure 4.2: Modern C Phrygian mode

The Meta-Event should be placed at time zero in the very first track of the MIDI file, before any Scale-Key Meta-Events.

4.1.8 Scale-Key Meta-Event

The Scale-Key Meta-Event declares what note the scale with the specified ID begins at. Its syntax is:

Scale-Key Meta-Event

```
0xFF + <Type-byte> + <Length-byte> + 0x65 + <Note-byte> +
     + <ID-byte>
```

HarmonicMIDI type-byte is set at 0x65. The note byte encodes a note in the same manner as in the Chord Section Meta-Event, with the first four bytes specifying the accidental and the last four the natural of the note. The ID-byte references a scale declared by a Scale Declaration Meta-Event. As in the HMIDI Key Signature Meta-Event, an empty Scale-Key Meta-Event can be declared, conveying an unassigned key.

4.2 Implementation

We have implemented the HarmonicMIDI Meta-Events as classes in the Java programming language. In addition, we have developed two command line programs in Java as a proof-of-concept of the encoding. The first one uses the results of the harmonic analysis algorithm described in the previous chapter to extend the related MIDI file with the corresponding HarmonicMIDI Meta-Events. The second program reads an extended MIDI file and decodes the harmonic information it contains.

Conclusion

We have analyzed and compared numerous musical codes — assessing their strengths and weaknesses. Doing so, we have come to the conclusion that the Standard MIDI File format suits our needs best. We introduce the HarmonicMIDI extension, which allows the user to encode harmonic analysis data within .mid files. It retains complete compatibility with the Standard MIDI File format. The extended .mid files remain compact and easily parseable. We have developed two Java programs to verify the viability of the new encoding.

However, before the HarmonicMIDI extension can enjoy practical use, more work is required in developing software tools for writing and reading the extended format. As an example, a plug-in for a scorewriting program, such as Sibelius or MuseScore, which would allow users to add harmonic information to the their compositions and read the harmonic data present in extended .mid files in a user-friendly way.

If need be, the HarmonicMIDI extension can be easily expanded with more Meta-Events in the future; however, we believe the current form of the standard is more than sufficient for the purpose of storing harmonic analysis data.

Bibliography

[1] Csound site. https://csound.com/. Accessed: 2023-12-17.

[2] Dflat site. https://dflat.procedural.design/. Accessed: 2023-12-18.

[3] kunstderfuge.com. https://www.kunstderfuge.com/. Accessed: 2024-04-25.

[4] *MIDI 1.0 Detailed Specification*. The MIDI Manufacturers Association, 1996.

[5] J. Murray Barbour. The Principles of Greek Notation. *Journal of the American Musicological Society*, 13(1):1–17, 1960.

[6] Indiana University Bloomington. Introduction to the MIDI Standard. https://cecm.indiana.edu/361/midi.html. Accessed: 2024-03-04.

[7] Encylopedia Britannica. Cadence. https://www.britannica.com/art/cadence-music. Accessed: 2024-05-01.

[8] Encylopedia Britannica. Evolution of Western staff notation. https://www.britannica.com/art/musical-notation/Evolution-of-Western-staff-notation. Accessed: 2023-12-18.

[9] CTAN. MusiXTeX. https://ctan.org/pkg/musixtex. Accessed: 2024-03-26.

[10] Raymond Erickson. "The Darms Project": A Status Report. *Computers and the Humanities*, 17(9):291–298, 1975.

[11] Raymond Erickson. Musicomp 76 and the State of DARMS. *College Music Symposium*, 17(1):90–101, 1977.

[12] Eva Ferková, Silvia Adamová, Michal Šukola, and Hana Urbancová. Computer search tool for harmonic structures and progressions in MIDI files and possible applications. *Clavibus unitis*, 9(3):1–14, 2020.

[13] W3C Music Notation Community Group. Final Community Group Report 1 June 2021. https://www.w3.org/2021/06/musicxml40/. Accessed: 2024-03-26.

[14] Goffredo Haus. Notes About MIDI Specifications. https://www.lim.di.unimi.it/IEEE/MIDI/INDEX.HTM. Accessed: 2024-03-17.

[15] Wayne Howard. Music and Accentuation in Vedic Literature. *The World of Music*, 24(3):23–34, 1982.

[16] Music Encoding Initiative. An introduction to MEI. https://music-encoding.org/about/#. Accessed: 2024-05-04.

[17] Music Encoding Initiative. MEI guidelines (5.0). https://music-encoding.org/guidelines/v5/content/. Accessed: 2024-05-06.

[18] A. D. Kilmer and M. Civil. Old Babylonian Musical Instructions Relating to Hymnody. *Journal of Cuneiform Studies*, 38(1):94–98, 1986.

[19] Cornelius C. Noack. Typesetting music with PMX. http://icking-music-archive.org/software/pmx/pmxccn.pdf. Accessed: 2024-05-09.

[20] Library of Congress. The Sustainability of Digital Formats — MusicXML, Version 3.1. https://www.loc.gov/preservation/digital/formats/fdd/fdd000499.shtml. Accessed: 2024-03-26.

[21] Craig Stuart Sapp. Standard MIDI File Structure. https://ccrma.stanford.edu/~craig/14q/midifile/MidiFileFormat.html. Accessed: 2024-03-19.

[22] Eleanor Selfridge-Field. *Beyond MIDI: The Handbook of Musical Codes*. 1997.

[23] MusicXML Site. Software support. https://www.musicxml.com/software/. Accessed: 2024-03-26.

[24] W3C. Extensible Markup Language (XML) 1.0 (Fifth Edition). https://www.w3.org/TR/REC-xml/. Accessed: 2024-05-04.

Appendix A: Digital material

All programs mentioned in the thesis, along with the other supplementary material, is included in the electronic attachment. The latest version of the programs can be found here: `https://github.com/tfakult35/HarmonicMIDIExtension`. A demo of the algorithmic harmonic analysis program can be found on this site: `https://analysisofharmony.sk/demo.php`.