MIDI Programming in Scheme

Supported by an Emacs Environment

Kurt Nørmark
Department of Computer Science
Aalborg University
Denmark
normark@cs.aau.dk

ABSTRACT

A Scheme representation of Standard MIDI Files is proposed. The Scheme expressions are defined and constrained by an XML-language, which in the starting point is inspired by a MIDI XML event language made by the MIDI Manufactures Association. The representation of Standard MIDI Files in Scheme makes it possible to carry out systematic modifications and transformations of MIDI contents with use of pure functional programming. Side by side with the XML-inspired MIDI language, the paper describes an Emacs-based, textual programming environment that supports the MIDI programming process. The programming environment also supports a variety of interactive features - similar to MIDI sequencers - but restricted to a textual representation of the music. The main contributions of the work are considered to be (1) An accumulated MIDI function library, which can transform MIDI files in many non-trivial ways; (2) A proposed working process alternating between creative mode and programmatic editing mode within a MIDI programming environment; and (3) A textual MIDI programming environment with embedded support of many interactive, MIDI-related functionalities.

Categories and Subject Descriptors

D.1.1 [**Applicative (Functional) Programming**]: Lisp, Scheme; H.5.5 [**Sound and Music Computing**]: Systems; I.7.2 [**Document Preparation**]: Markup Languages

1. INTRODUCTION

This paper is about MIDI programming in the functional programming paradigm [11]. More specifically, MIDI programs are expressed in Scheme [12], which is a language in the Lisp [19] family. As such, the paper deals with a programmatic approach to creation and modification of sequences of MIDI messages via use of *pure functions*. The paper specializes to a situation where a MIDI sequence is represented as a piece of program - rendered from a MIDI file, or authored in the Scheme programming language. The practical handling of MIDI programs in Scheme is supported by a comprehensive MIDI programming environment, programmed in Emacs Lisp, for use in the GNU Emacs text editor [18]. The proposed music representation language is conceptually close to the MIDI language. The MIDI music language is defined and constrained as an XML language [22] which is brought into the programming language via LAML [14] (a software package programmed by the author of this paper).

There exists a very large and diverse amount of software for handling of music. Each such piece of software is envisioned to support a musician in some specific way, and in a specific application context. The application context of the software described in this paper is the following:

> *The music is basically created and captured via a MIDI instrument (a keyboard). The music we have in mind is mainly traditional, western popular music. The MIDI programming facilities are supposed to facilitate systematic modification of the results recorded on the MIDI instrument. Typically, a song is transferred back and forth from the keyboard to the textual programming environment several times before it is finished.*

The software described in this paper is geared towards programmatic manipulation of Standard MIDI files (format zero and one). It seems to be the case that no fixed set of existing tools is sufficient for editing of MIDI files. In other words, there seems to be an *endless need* of modifications of MIDI contents, each of which calls for a new piece of effectuating program. Our work is based on a general-purpose, high-level programming language which represents and accesses the MIDI information in a flexible way. Our approach is flexible enough for relatively easy and smooth development of the necessary transformation functions, and efficient enough for handling of real music.

The paper contributes in the following areas:

1. Representation of standard MIDI files as expressions in a functional programming language, as constrained and defined by an XML language (by way of an XML Document Type Definition - a DTD).
2. Programmatic augmentation of MIDI files for the sake

of doing systematic transformations of MIDI event sequences.

3. Programmatic creation of small MIDI pieces which can be added as constituents of more complex MIDI files.

4. A textual, operational environment which supports the MIDI programming process. In addition, the environment supports interactive editing operations on the textually represented MIDI events.

A paper like this can be written from a computer science perspective, from a music perspective, or from a combined perspective. This paper is angled from the area of computer science (especially from interests in functional programming in Lisp languages). The paper describes the work - and the results - of uniting a professional approach to programming with a leisure approach to production of music.

2. BACKGROUND

In this section we will - in a concise way - provide background information on technologies and languages that are important to our work.

2.1 MIDI

MIDI[1] (Musical Instrument Digital Interface) is a *protocol* for exchange of musical events. MIDI emphasizes the *sequencing of messages* - not the audio contents as such. The most central messages are `NoteOn` (characterized by note value, the assigned channel, and a velocity) and `NoteOff` which, respectively, denote the start and the end of a note. In addition, a variety of other messages (such as messages for selection of instruments, key pressure, volume, panning, and pitch bend) control different aspects of the music.

The MIDI contents - a piece of music - can be represented in a compact, binary format called *standard midi files*. A format 0 standard midi file consists of a header and a single track. A format 1 standard midi file may have more than one track. Each message in a standard midi file has a time stamp which denotes the delta time relative to the previous message in the track (or to the beginning of the track if no previous message exists).

In this work, we have developed an alternative representation of standard MIDI files in terms of expressions from a functional programming language, and closely related to a MIDI XML language.

2.2 MIDI Programming

The term "MIDI Programming" is used with several different meanings. In a loose sense, the term is used for activities that manipulate the individual parameters of MIDI messages (as a contrast to manipulating renderings at a higher level of abstraction). In a more strict interpretation, the term is used for writing programs, with use of a programming language, which manipulate a collection of MIDI messages. In this paper we use the latter interpretation.

We identify two different working modes when a musician deals with a piece of MIDI music. The first of these is *creative mode*, where the musician plays a piece of music using an electronic instrument. The performance of the musician is captured as a (long) linear sequence of MIDI messages. The other mode is *editing mode* where the musician modifies or creates single or multiple MIDI messages in a MIDI Sequencing Tool - on the instrument or on a computer. In principle, any piece of music can be created in editing mode, but the process is foreign to most musicians, and the outcome tends to be monotonous and "machine-like".

In editing mode it is possible to create or modify MIDI messages individually. In some situations this is sufficient, for instance for corrections of minor playing errors. In other situations, it is very difficult to reach the desired result by means of editing of individual MIDI messages. In order to obtain the desired modifications it is typically necessary to create or modify hundreds or thousands of messages. Needless to say, it is almost impossible to carry out such a complicated process manually without loosing control, or without ruining the music.

As a consequence of these observations it is desirable that the editing mode supports creation and modification at a more coarse grained level than that of the individual MIDI messages. A given MIDI sequencer may (or may not) support a fixed number of such editing tasks. The most powerful and complete solution in editing mode is, however, to allow for systematic manipulation of MIDI contents via a full-fledged (and Turing complete) programming language. In such a setup, it is possible to prescribe any systematic modification of a piece of music via a piece of program, which is activated on the MIDI contents. This will be the meaning of *MIDI programming* in this paper.

In systems such as Haskore [10] the source of the music is a Haskell expression at a relatively high abstraction level. When the expression is executed, an equivalent low-level representation, such as a MIDI sequence, is derived. When an editing need arises, the high-level representation of the music is modified. As already mentioned in the introduction, the work proposed in this paper supports another working process. The music is, in the starting point, captured (recorded) as a sequence of MIDI messages using an electronic MIDI instrument in creative mode. The MIDI program manipulates a representation of the music, which is relative close to MIDI level (in editing mode, at a computer external to the MIDI instrument). Typically, the music is passed back and forth between the instrument and the programming environment. This working process prevents solutions which raises the level of abstraction to a level much higher than that of MIDI. The intended working process is illustrated in Figure 1.

2.3 Functional Programming and Scheme

A functional program consists of an *expression* which can be evaluated to a *value*. The evaluation of the expression has no effect on any variable, and it has no side-effects (such as

[1]The official documentation of the MIDI protocol and Standard MIDI Files comes from the MIDI Manufacturers Organization, `www.midi.org`. Unfortunately, the standard documents are not freely available on the internet, and they are distributed in print only. Fortunately, there exists excellent alternative descriptions of MIDI on the internet, such as the resources located at `http://home.roadrunner.com/~jgglatt/`.

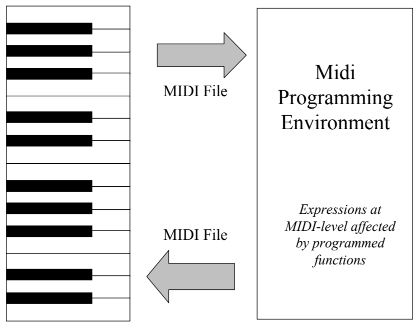
MIDI File
Midi Programming Environment
Expressions at MIDI-level affected by programmed functions
MIDI File

Figure 1: *An iterative working process alternating between creative mode at the left and editing mode at the right.*

output to the screen, or to files). Functions - the primary abstractions - are *pure mappings* of input to output. As we will see in Section 2.2 of this paper, we will represent a song as a single expression. The value of the expression can be thought of as a sequence of MIDI messages. (In reality the value of the expression is an intermediate tree structure, from which the MIDI messages can be derived). Functions may transform one or more MIDI messages to sequences of other MIDI messages - without affecting or mutating the input messages as such. Iteration is governed by recursive processing of composite data structures (typically lists). Recursive patterns, such as mapping and filtering functions, are captured via higher-order functions - functions which accept functions as parameters and/or return functions as results.

Functional programming stands as a contrast to imperative programming, which is characterized by assignments to variables, side-effects, use of iterative control structures, and procedures that mutate the state of the program. Functional programming is usually considered as programming at a higher level, as more abstract, and mathematically more clean than imperative programming.

Scheme [12] - a Lisp dialect - is a relatively small, powerful, and elegant language for list processing. Syntactically, a Scheme program is a nested and fully parenthesized list structure. In addition, lists are the primary data structure supported by Scheme. Besides a strong support of functional programming, Scheme also supports imperative programming, and it is strong enough to embrace some of the key ideas in object-oriented programming.

2.4 XML and LAML

XML [22] is a meta language, and as such it can be used to establish new languages that share a common "HTML-like syntactical basis". XML is the meta language behind important WEB languages such as XHTML, SVG, and XSL. The grammar of an XML language (described in either a DTD or an XML Schema) prescribes a number of *elements*, and a number of *attributes* per element.

It is possible to rephrase many existing languages as XML languages. MIDI is no exception. The MIDI Manufacturers Association have defined XML languages[2] for various aspects of the MIDI standard, of which the MIDI XML event language has been used as the starting point of our work. MusicXML [6] is another XML format for music, primarily intended as an interchange format between computer music tools. Figure 2 shows a small excerpt of some MIDI messages represented in a MIDI XML language.

LAML [14] offers a systematic way to embed an XML language in Scheme. Each element of the XML language is mapped to a function in Scheme. With use of LAML, it is natural to integrate the declarative and descriptive aspects of XML with the processing power of a full-fledged, functional programming language. Thus, with use of LAML, the expressiveness of the programming language is available at any point in the XML document, and at any time of the development process that leads to the final document. In conventional XML documents, such power must be brought in *from the outside* via tools that process the XML document. By way of LAML, an XML language is represented as a library of *mirror functions* in Scheme, and an XML document is represented as a Scheme expression which applies these functions. In this setup, the processing power comes *from inside* the document itself. The mirror functions generate an internal representation of the XML document (an abstract syntax tree). In addition, the mirror function can validate the XML document relative to the grammar (DTD) when they are called.

LAML has been used to handle and support several mainstream XML languages in Scheme. This includes XHTML and SVG [16]. In addition, a number of new XML languages have been defined and turned into Scheme libraries, for instance the lecture note language called LENO [15]. Figure 3 shows the LAML representation of the XML fragment from Figure 2. This paper describes how Scheme and LAML are used for transformation and editing of MIDI files via programmatic solutions.

3. MIDI PROGRAMMING IN SCHEME

In this section we will describe how we support a functional approach to MIDI programming in Scheme. Together with Section 4 this is the main part of the paper.

A MIDI file can be converted to a MIDI LAML file. This conversion consists of a *parsing* process of the binary MIDI data, followed by a *transformation* process to either absolute time or delta time mode (see Section 3.1 below). Figure 3 shows a delta time MIDI LAML file with a sequence of notes from C3 to C4. A MIDI LAML file is represented as a Scheme expression - and as such it is a (functional) program. Most MIDI LAML files, which represent a real piece of music, are much longer (typically 1 MB of text, corresponding to a few thousand MIDI events). Most of the attributes (such as `note`, `channel`, `velocity`) have direct MIDI counterparts.

The `StandardMidiFile` form is the root of a single Scheme expression. The value of this expression - an abstract syntax tree - represents the song as such. The abstract syntax tree can be rendered in various ways by the MIDI program-

[2]http://www.midi.org/dtds/midi_xml.php

```
<StandardMidiFile>
   <MidiHeader format="0" numberOfTracks="1" pulsesPerQuarterNote="480" />
   <MidiTrack>
      <ControlChange deltaTime="0" channel="1" control="0" value="0" />
      <ControlChange deltaTime="0" channel="1" control="32" value="0" />
      <ProgramChange deltaTime="0" channel="1" number="1" />
      <Meta deltaTime="0" type="81"> 0B 71 B0</Meta>
      <NoteOn deltaTime="480" channel="1" note="60" velocity="88" duration="202" />
      <NoteOn deltaTime="119" channel="1" note="61" velocity="114" duration="292" />
      <NoteOn deltaTime="131" channel="1" note="62" velocity="87" duration="201" />
      <NoteOn deltaTime="119" channel="1" note="63" velocity="114" duration="290" />
      <NoteOn deltaTime="131" channel="1" note="64" velocity="114" duration="290" />
      <NoteOn deltaTime="131" channel="1" note="65" velocity="87" duration="199" />
      <NoteOn deltaTime="119" channel="1" note="66" velocity="114" duration="289" />
      <NoteOn deltaTime="130" channel="1" note="67" velocity="113" duration="288" />
      <NoteOn deltaTime="130" channel="1" note="68" velocity="86" duration="197" />
      <NoteOn deltaTime="118" channel="1" note="69" velocity="86" duration="196" />
      <NoteOn deltaTime="118" channel="1" note="70" velocity="86" duration="196" />
      <NoteOn deltaTime="118" channel="1" note="71" velocity="85" duration="195" />
      <NoteOn deltaTime="118" channel="1" note="72" velocity="85" duration="194" />
      <Meta deltaTime="1920" type="0"></Meta>
   </MidiTrack>
</StandardMidiFile>
```

Figure 2: *A Standard Midi file represented in a MIDI XML language.*

```
(StandardMidiFile
  (MidiHeader 'format "0" 'numberOfTracks "1" 'pulsesPerQuarterNote "480" 'mode "deltaTime")
  (MidiTrack
     (ControlChange 'deltaTime "0" 'info "0:0:0 Bank select MSB" 'channel "1" 'control "0" 'value "0")
     (ControlChange 'deltaTime "0" 'info "0:0:0 Bank select LSB" 'channel "1" 'control "32" 'value "0")
     (ProgramChange 'deltaTime "0" 'info "0:0:0 **GM Piano: Bright Acoustic Piano" 'channel "1" 'number "1")
     (Meta 'deltaTime "0" 'info "0:0:0 Tempo: 80 BPM." 'type "81" "0B 71 B0")
     (NoteOn 'deltaTime "480" 'info "0:1:0 C3" 'channel "1" 'note "60" 'velocity "88" 'duration "202")
     (NoteOn 'deltaTime "119" 'info "0:1:119 C#3" 'channel "1" 'note "61" 'velocity "114" 'duration "292")
     (NoteOn 'deltaTime "131" 'info "0:1:250 D3" 'channel "1" 'note "62" 'velocity "87" 'duration "201")
     (NoteOn 'deltaTime "119" 'info "0:1:369 D#3" 'channel "1" 'note "63" 'velocity "114" 'duration "290")
     (NoteOn 'deltaTime "131" 'info "0:2:20 E3" 'channel "1" 'note "64" 'velocity "114" 'duration "290")
     (NoteOn 'deltaTime "131" 'info "0:2:151 F3" 'channel "1" 'note "65" 'velocity "87" 'duration "199")
     (NoteOn 'deltaTime "119" 'info "0:2:270 F#3" 'channel "1" 'note "66" 'velocity "114" 'duration "289")
     (NoteOn 'deltaTime "130" 'info "0:2:400 G3" 'channel "1" 'note "67" 'velocity "113" 'duration "288")
     (NoteOn 'deltaTime "130" 'info "0:3:50 G#3" 'channel "1" 'note "68" 'velocity "86" 'duration "197")
     (NoteOn 'deltaTime "118" 'info "0:3:168 A3" 'channel "1" 'note "69" 'velocity "86" 'duration "196")
     (NoteOn 'deltaTime "118" 'info "0:3:286 A#3" 'channel "1" 'note "70" 'velocity "86" 'duration "196")
     (NoteOn 'deltaTime "118" 'info "0:3:404 B3" 'channel "1" 'note "71" 'velocity "85" 'duration "195")
     (NoteOn 'deltaTime "118" 'info "1:0:42 C4" 'channel "1" 'note "72" 'velocity "85" 'duration "194")
     (Meta 'deltaTime "1920" 'info "2:0:42 Sequence number" 'type "0")
  )
)
```

Figure 3: *A Standard MIDI file represented as a Scheme and LAML expression. The info attributes (presentation only) are intended to aid the human understanding of individual MIDI messages. In this example, the info attributes decode and explain the* `deltaTime` *and* `note` *attributes.*

ming environment (see Section 4), for instance as a MIDI file which will be played upon evaluation of the expression. A so-called *action procedure* of the top-level `StandardMidiFile` form grasps the AST, and delivers it to some kind of additional processing. The default action (under control of the MIDI programming environment) is to play the music.

3.1 Time Modes

In Standard MIDI files, each event is timed relative to the previous event (or to the start of the track if the event is the first in a sequence). This timing mode, called *deltaTime*, is supported in MIDI LAML. In addition we support an *absTime* mode, where each event is timed relative to the beginning of a song. The mode is established as an attribute in the `MidiHeader`. In both modes, a `NoteOn` event is implicitly paired with the corresponding MIDI `NoteOff` event by introduction of an XML `duration` attribute.

In absTime mode it is possible to insert a sequence of deltaTimed midi events, which will be timed relative to the previous event in absTime mode. AbsTime phrases are good for entire and completed songs, which under normal circumstances should keep their rhythmic structure. DeltaTimed phrases are useful for smaller pieces of music (phrases) which eventually should be inserted into, or derived to, a song in absTime mode. The operational environment (see Section 4) allows for smooth shifting between the two time modes.

A *normalized MIDI LAML file* is expressed entirely with use of the XML mirror functions of the MIDI XML language. In absTime mode, a normalized MIDI LAML file additionally maintains the invariant that the messages in a MIDI track have non-decreasing time values. The program in Figure 3 is normalized; The programs in the Figure 4 and Figure 6 are not, because they call (non-mirror) functions from the MIDI function library.

3.2 The Function Library

During the last couple of years we have developed a library of Scheme functions which implement many useful transformations of MIDI music. But as noticed already in the introduction, new needs seem to appear over and over, and therefore it is important that the system allows for easy programming of new transformations. The most useful and versatile of such functions are organized and documented in the MIDI LAML library for convenient reuse.

The functions in the existing MIDI LAML library can be categorized (non-exhaustively) in the following way:

1. MIDI message *list functions* which transform a list of MIDI messages.
2. MIDI message *factory functions* that generate a single midi message (or a small, constant number of MIDI messages).
3. Functions that address and transform given bars or given named sections of a MIDL LAML file.
4. Functions that transform an entire standard MIDI file.
5. Predicates which identify a certain subset of MIDI messages.
6. Functions that extract attributes of a single MIDI event.
7. Channel-related functions that copy, join, or delete certain messages that belong to a given channel.

The MIDI message list functions of category 1 are the "bread and butter functions" of the library. We illustrate applications of two such functions in Figure 4, namely `quantize` and `scale-attribute`. The `quantize` function regularizes the value of the `deltaTime` attribute to 16'th notes (corresponding to values such as 0, 120, 240, 360, 480, etc. for songs with the resolution of 480 pulses per quarter note). The `scale-attribute` function, as applied in Figure 4, scales the values of the `velocity` attributes with a function constructed by `make-scale-function-by-xy-points`. The scaling function, shown in Figure 5, scales the velocities of the affected `NoteOn` messages from (113, 86, 86, 86, 85) to (102, 69, 53, 37, 21). The domain of all scaling functions is the real numbers in the interval [0...1]. Both `quantize` and `scale-attribute` return a list of transformed notes, which together with other contextual notes are passed as parameters to the `MidiTrack` function. By way of the LAML parameter passing rules [14] these lists are (recursively) spliced together to a single flat list. The automatic "flattening" of XML elements, done by LAML is a major, is a major practical asset.

In Figure 4 the function `quantize` is applied on one region of MIDI content, say *r1*, and the function `scale-attribute` is applied on another region, say *r2*. In figure 4 the regions *r1* and *r2* are disjoint. A subtle *overlapping regions problem* occurs if the regions *r1* and *r2* overlaps - without one of them being included in the other. It would - in general - be awkward to program a functional solution to this problem. We outline an "environmental solution" when we discuss the MIDI LAML stack in Section 4.

On some occasions, the value of an attribute in one MIDI message should depend on attributes in contextual messages. As an example, we may want one `NoteOn` message to last (via adjustment of the `duration` attribute) until some other specific `NoteOn` message starts. In simulating guitar playing, we may want the sound generated from touching a given string to last until the string is touched again. Such contextual dependencies must necessarily by resolved in a two-phased evaluation process, because the actual evaluation order in a functional program execution is unknown. In the first phase, the entire abstract syntax tree must be established, and in the second phase the missing attributes must be calculated based on some contextual tree traversal.

In LAML, one or more missing attributes (or content elements) can be represented by a so-called *delayed procedural content item*. A delayed procedural content item is a function, which is stored as part of the AST, and which contains an expression that calculates the context-dependent information from given parameters. The function is called automatically in a second evaluation phase. Upon activation of the function, the nearest enclosing AST node and the AST document root node are passed as actual parameters. The function is supposed to return a list of attribute/value pairs. Figure 6 illustrates the idea by a simple concrete example. The expression `(duration-to-next d)` returns a function, which measures the duration to the next similar `NoteOn` message, or `d` in case no such message exists. The

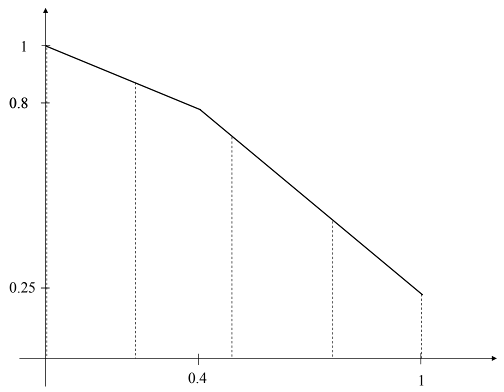

Figure 5: *The scaling function used in Figure 4. It is produced by the expression* `(make-scale-function-by-xy-points (from-percent-points '((0 100) (40 80) (100 25))))`.

```
(StandardMidiFile
  (MidiHeader 'format "0" 'numberOfTracks "1" 'pulsesPerQuarterNote "480" 'mode "deltaTime")
  (MidiTrack
    (ControlChange 'deltaTime "0" 'info "0:0:0 Bank select MSB" 'channel "1" 'control "0" 'value "0")
    (ControlChange 'deltaTime "0" 'info "0:0:0 Bank select LSB" 'channel "1" 'control "32" 'value "0")
    (ProgramChange 'deltaTime "0" 'info "0:0:0 **GM Piano: Bright Acoustic Piano" 'channel "1" 'number "1")
    (Meta 'deltaTime "0" 'info "0:0:0 Tempo 80 BPM." 'type "81" "0B 71 B0")
    (NoteOn 'deltaTime "480" 'info "0:1:0 C3" 'channel "1" 'note "60" 'velocity "88" 'duration "202")
    (quantize 1 16 480
      (NoteOn 'deltaTime "119" 'info "0:1:119 C#3" 'channel "1" 'note "61" 'velocity "114" 'duration "292")  ; r1:
      (NoteOn 'deltaTime "131" 'info "0:1:250 D3" 'channel "1" 'note "62" 'velocity "87" 'duration "201")
      (NoteOn 'deltaTime "119" 'info "0:1:369 D#3" 'channel "1" 'note "63" 'velocity "114" 'duration "290")
      (NoteOn 'deltaTime "131" 'info "0:2:20 E3" 'channel "1" 'note "64" 'velocity "114" 'duration "290")
      (NoteOn 'deltaTime "131" 'info "0:2:151 F3" 'channel "1" 'note "65" 'velocity "87" 'duration "199")
    )
    (NoteOn 'deltaTime "119" 'info "0:2:270 F#3" 'channel "1" 'note "66" 'velocity "114" 'duration "289")
    (scale-attribute 'velocity
      (make-scale-function-by-xy-points (from-percent-points '((0 100) (40 80) (100 25))))
       (NoteOn 'deltaTime "130" 'info "0:2:400 G3" 'channel "1" 'note "67" 'velocity "113" 'duration "288")  ; r2:
       (NoteOn 'deltaTime "130" 'info "0:3:50 G#3" 'channel "1" 'note "68" 'velocity "86" 'duration "197")
       (NoteOn 'deltaTime "118" 'info "0:3:168 A3" 'channel "1" 'note "69" 'velocity "86" 'duration "196")
       (NoteOn 'deltaTime "118" 'info "0:3:286 A#3" 'channel "1" 'note "70" 'velocity "86" 'duration "196")
       (NoteOn 'deltaTime "118" 'info "0:3:404 B3" 'channel "1" 'note "71" 'velocity "85" 'duration "195")
    )
    (NoteOn 'deltaTime "118" 'info "1:0:42 C4" 'channel "1" 'note "72" 'velocity "85" 'duration "194")
    (Meta 'deltaTime "1920" 'info "2:0:42 Sequence number" 'type "0")
  )
)
```

Figure 4: *Sample use of two MIDI message lists functions,* `quantize` *and* `scale-attribute`*, on two disjoint regions (r1 and r2) of MIDI messages.*

function `duration-to-next` is part of the MIDI LAML library.

3.3 Typical Tasks

The previous section described technical aspects of the existing function library. In this section we will enumerate a number of typical musical transformation tasks that we have carried out by use of functions from the library.

1. Quantification or randomization of notes to/from standard units with the purpose of either constraining or loosening of the timing of a piece of music.

2. Systematic modification of a selected MIDI event attribute. This may, for instance, be stretching or displacing the time attributes, or scaling of the velocity attribute. Both scaling by constant factors, and scaling by use of scaling functions (such as shown in Figure 5) are supported.

3. Joining events in two channels to a single channel, or (more challenging), splitting the events of a single channels to two channels. The channel splitting can be controlled by a predicate, which may be allowed to exercise the context of a candidate MIDI message.

4. Generation of pitch bend change lists (a list of PitchBendChange MIDI messages) from a scaling function with a range of [-1 .. 1].

5. Addition of drum fills. The drum fills are added by substituting designated bars in given channels (containing the ordinary drum phrase) with MIDI phrases that interrupts or breaks the ordinary drum rhythm. It may be tricky to arrange for such variations interactively while playing. It is even more difficult to add them by simple, interactive editing of the MIDI event list.

6. Automatic introduction of markers (see Section 3.4) based on an analysis of a song. The analysis may be based on the density of `NoteOn` messages in a given (and fixed) period of time.

Some of the tasks mentioned above are widely supported by hardware or software sequencers. Most of the tasks, however, are so specialized that it only is realistic to support them through a piece of program, written in a full-fledged programming language. This observation is the *raison d'etre* of the MIDI LAML system.

3.4 Imposing Structure on MIDI Contents

In contrast to most hand-written Scheme programs, a MIDI LAML program is poor with respect to *structure*. A long sequence of MIDI event is in the starting point automatically turned into a long list of XML elements of a MIDI LAML program. The introduction of a better structure is a key - and in many cases a prerequisite - to making interesting and useful programmatic changes to a song. Without such a structure it is difficult to navigate the song, and it is difficult to identify the MIDI events which we want to transform. The following structuring mechanisms are provided for MIDI LAML:

- Use of MIDI format 1, which divides the MIDI contents

```
(StandardMidiFile
  (MidiHeader 'format "0" 'numberOfTracks "1" 'pulsesPerQuarterNote "480" 'mode "deltaTime")
  (MidiTrack
    (ControlChange 'deltaTime "0" 'info "0:0:0 Bank select MSB" 'channel "1" 'control "0" 'value "0")
    (ControlChange 'deltaTime "0" 'info "0:0:0 Bank select LSB" 'channel "1" 'control "32" 'value "0")
    (ProgramChange 'deltaTime "0" 'info "0:0:0 **GM Piano: Bright Acoustic Piano" 'channel "1" 'number "1")
    (Meta 'deltaTime "0" 'info "0:0:0 Tempo 20 BPM." 'type "81" "2D C6 C0")
    (NoteOn 'deltaTime "480" 'info "0:1:0 C3" 'channel "1" 'note "60" 'velocity "88" (duration-to-next 30))
    (NoteOn 'deltaTime "120" 'info "0:1:120 E3" 'channel "1" 'note "64" 'velocity "114" 'duration "30")
    (NoteOn 'deltaTime "120" 'info "0:1:240 G3" 'channel "1" 'note "67" 'velocity "113" 'duration "30")
    (NoteOn 'deltaTime "120" 'info "0:1:360 C3" 'channel "1" 'note "60" 'velocity "88"  (duration-to-next 30))
    (NoteOn 'deltaTime "120" 'info "0:2:0 E3" 'channel "1" 'note "64" 'velocity "114" 'duration "30")
    (NoteOn 'deltaTime "120" 'info "0:2:120 G3" 'channel "1" 'note "67" 'velocity "113" 'duration "30")
    (NoteOn 'deltaTime "120" 'info "0:2:240 C3" 'channel "1" 'note "60" 'velocity "88"  (duration-to-next 30))
    (NoteOn 'deltaTime "120" 'info "0:2:360 E3" 'channel "1" 'note "64" 'velocity "114" 'duration "30")
    (NoteOn 'deltaTime "120" 'info "0:3:0 G3" 'channel "1" 'note "67" 'velocity "113" 'duration "30")
    (Meta 'deltaTime "1920" 'info "1:3:0 Sequence number" 'type "0")
  )
)
```

Figure 6: *A few C3-E3-G3 notes, where all C3 notes are requested to play until the next C3 note (or 30 ticks if no C3 follows)."*

into tracks.

- Use of the delimitations of bars and beats in the MIDI contents.
- Introduction of *markers* (represented as meta events), and, by means of these, *named sections* of the MIDI contents.

The bar structure is visualized by the `info` attributes, which can be seen in Figures 3, 4, and 6. In addition, the environment makes it possible to visualize the bars explicitly in the source program by means of distinguished program comments, called *bar comments*, see Figure 7.

The use of Meta events allows us to mark, and to name constituents of a song, be it parts (such as A and B parts of a song), verses, or song lines. This is very useful, because many transformations address such substructures of a song. The programming environment supports a *marker browser*, which provides an overview of a MIDI LAML song. In addition, the environment prescribes a specific technique for the introducing and inserting markers into a song. In Section 4 we will describe how this is accomplished.

4. A MIDI ENVIRONMENT

In this section we will describe the practical use of the MIDI LAML language in a MIDI Programming Environment. The environment includes a number of interactive features which makes it possible to listen to selected parts of the music (ranging from an entire song to a single note).

MIDI LAML programs can, in principle, be handled from an interactive shell - in the style of the UNIX environment. We have, alternatively, decided to handle MIDI LAML programs from a text editing environment, namely GNU Emacs [18]. This turns out to be flexible, powerful, and natural because a MIDI LAML program is indeed a *textual representation* of MIDI contents. The Emacs text editor provides easy access to the operations of the environment via pull-down menus. Most operations in the MIDI LAML environment are organized in such menus. In addition, the operations in the environment can be reached via keyboard shortcuts and/or function keys.

4.1 Basic MIDI Programming Support

The primary aim of the environment is to support the programmatic refinement of a piece of music. As described in Section 2.2, the starting point is a MIDI LAML file, like the one shown in Figure 3, typically parsed up from a standard MIDI file. The programmer inserts pieces of programs such as shown in Figure 4 and 6. Typically, the programmer calls one or more functions from the MIDI LAML library on selected sequences of MIDI events. During this process the programmer is supported by an interactive *help system*, and by automatic *completion of names.* Smooth help during programming - including name completion - can be activated on any XML form, on any XML attribute, and on any function from the library, just by hitting the TAB key. This kind of help is similar to intellisense, as used in Visual Studio.

As discussed in Section 3.4, the emphasis on the source program structure is crucial for the programmer. The environment can help emphasize the *bar structure* of a piece of music by inserting a distinguished source program comment in between every bar, see Figure 7. In addition, it is possible to ask for generation of *score comments* in the right-hand side of an editing window. A collection of score comments shows a (primitive) vertical rendering of the notes in sheet music form (shown vertically). This is also illustrated in Figure 7. The score comments are kept up-to-date if the music is edited interactively, see Section 4.4.

In principle, the source program may serve as the primary representation of the music during the rest of the music's life time. That would imply that any subsequent modification - programmatic or interactive - should take place in the source program, such as shown in Figures 3, 4, 6, and 7. This is not the approach we have taken. It is possible to normalize - and straighten out - the effect of the programmatic mod-

```
(StandardMidiFile
  (MidiHeader 'format "0" 'numberOfTracks "1" 'pulsesPerQuarterNote "480" 'mode "deltaTime")
  (MidiTrack
     (ControlChange 'deltaTime "0" 'info "0:0:0 Bank select MSB" 'channel "1" 'control "0" 'value "0")
     (ControlChange 'deltaTime "0" 'info "0:0:0 Bank select LSB" 'channel "1" 'control "32" 'value "0")
     (ProgramChange 'deltaTime "0" 'info "0:0:0 **GM Piano: Bright Acoustic Piano" 'channel "1" 'number "1")
     (Meta 'deltaTime "0" 'info "0:0:0 Tempo:20 BPM." 'type "81" "2D C6 C0")
     (NoteOn 'deltaTime "960" 'info "0:2:0 C3" 'channel "1" 'note "60" 'velocity "88" 'duration "600")            ; . . . o | | | | . . .
     (NoteOn 'deltaTime "240" 'info "0:2:240 E3" 'channel "1" 'note "64" 'velocity "114" 'duration "580")         ; . . . | o | | | . . .
     (NoteOn 'deltaTime "240" 'info "0:3:0 G3" 'channel "1" 'note "67" 'velocity "113" 'duration "576")           ; . . . | | o | | . . .
     (NoteOn 'deltaTime "240" 'info "0:3:240 C3" 'channel "1" 'note "60" 'velocity "88" 'duration "404")          ; . . . o | | | | . . .
; 1 --------------------------------------------------------------------------------------------------------
     (NoteOn 'deltaTime "240" 'info "1:0:0 E3" 'channel "1" 'note "64" 'velocity "114" 'duration "580")           ; . . . | o | | | . . .
     (NoteOn 'deltaTime "240" 'info "1:0:240 G3" 'channel "1" 'note "67" 'velocity "113" 'duration "576")         ; . . . | | o | | . . .
     (NoteOn 'deltaTime "240" 'info "1:1:0 C3" 'channel "1" 'note "60" 'velocity "88" 'duration "404")            ; . . . o | | | | . . .
     (NoteOn 'deltaTime "240" 'info "1:1:240 E3" 'channel "1" 'note "64" 'velocity "114" 'duration "580")         ; . . . | o | | | . . .
     (NoteOn 'deltaTime "240" 'info "1:2:0 G3" 'channel "1" 'note "67" 'velocity "113" 'duration "576")           ; . . . | | o | | . . .
     (NoteOn 'deltaTime "240" 'info "1:2:240 C3" 'channel "1" 'note "60" 'velocity "88" 'duration "600")          ; . . . o | | | | . . .
     (NoteOn 'deltaTime "240" 'info "1:3:0 E3" 'channel "1" 'note "64" 'velocity "114" 'duration "580")           ; . . . | o | | | . . .
     (NoteOn 'deltaTime "240" 'info "1:3:240 G3" 'channel "1" 'note "67" 'velocity "113" 'duration "576")         ; . . . | | o | | . . .
; 2 --------------------------------------------------------------------------------------------------------
     (NoteOn 'deltaTime "240" 'info "2:0:0 C3" 'channel "1" 'note "60" 'velocity "88" 'duration "404")            ; . . . o | | | | . . .
     (NoteOn 'deltaTime "240" 'info "2:0:240 E3" 'channel "1" 'note "64" 'velocity "114" 'duration "580")         ; . . . | o | | | . . .
     (NoteOn 'deltaTime "240" 'info "2:1:0 G3" 'channel "1" 'note "67" 'velocity "113" 'duration "576")           ; . . . | | o | | . . .
     (NoteOn 'deltaTime "240" 'info "2:1:240 C3" 'channel "1" 'note "60" 'velocity "88" 'duration "404")          ; . . . o | | | | . . .
     (NoteOn 'deltaTime "240" 'info "2:2:0 E3" 'channel "1" 'note "64" 'velocity "114" 'duration "580")           ; . . . | o | | | . . .
     (NoteOn 'deltaTime "240" 'info "2:2:240 G3" 'channel "1" 'note "67" 'velocity "113" 'duration "576")         ; . . . | | o | | . . .
     (NoteOn 'deltaTime "240" 'info "2:3:0 C3" 'channel "1" 'note "60" 'velocity "88" 'duration "600")            ; . . . o | | | | . . .
     (NoteOn 'deltaTime "240" 'info "2:3:240 E3" 'channel "1" 'note "64" 'velocity "114" 'duration "580")         ; . . . | o | | | . . .
; 3 --------------------------------------------------------------------------------------------------------
     (Meta 'deltaTime "480" 'info "3:0:240 Sequence number" 'type "0")
  )
)
```

Figure 7: *Illustration of bar comments (horizontal rules) and primitive vertical score comments (to the right). Both have the status of semicolon prefixed 'end-of-line comments' in Scheme.*

ifications by regenerating the underlying MIDI file, and by reestablishing a fresh MIDI LAML file that subsumes the modifications. (Normalization has been described in Section Section 3.1). Normalization is necessary in order to use most interactive features of the Emacs MIDI environment, see Section 4.4. Equally important, it is possible to bring a normalized MIDI LAML file out of the environment for external editing - for instance in creative mode of a MIDI instrument - and to bring it back again to programmatic editing. This working process is supported and organized via use of the MIDI LAML Stack, which is described next.

4.2 The MIDI LAML Stack

The MIDI LAML Stack is a stack of MIDI LAML source programs. When a new entry is pushed on the stack, the MIDI LAML program is evaluated, rendered as a MIDI file, and re-processed from the MIDI file to a new and fresh MIDI LAML file which is pushed on the stack. Such a push operation normalizes the MIDI LAML program, in the sense defined in Section 3.1.

The MIDI LAML stack offers an "environmental solution" to the overlapping region problem, which we introduced in Section 3.2. Using the MIDI LAML stack, the effect of the first transformation is programmed on the first region (*r1*), and a normalized a MIDI LAML program is pushed onto the stack. On the normalized MIDI LAML program, the second transformation can be carried out on the second region (*r2*). Thus, the two transformations on the overlapping regions *r1* and *r2* are carried out in two *evaluation stages* which are organized and kept together on the stack.

A number of push variations exist. In one variation, a transition is taken from absTime to deltaTime mode (or vice versa). In another, it is possible just to push a replication of a stack entry to the top of the stack. An external MIDI file (for instance brought back from an instrument) can also be pushed onto the stack. In that way, the MIDI LAML stack is used to keep a evolution of MIDI file versions together in a single unit.

When the stack is popped, the discarded stack entry becomes a *ghost entry*. Ghost entries can be recovered by the **pop undo** operation. With this organization it is possible reel back and forth through the stack entries. The ghost entries are lost, however, if a push operation is carried out, without first undoing pop operations of ghost entries. The loss of ghost elements may be a problem, because in a staged evaluation process, several valuable program contributions may occur exclusively in the ghost entries of the stack. Therefore, just before a push operation, the *essential program contributions* can be extracted from the ghost entries. These contributions are made available to the programmer in a special *ghost fragment browser*.

4.3 The Section Browser

In Section 3.4 we discussed the importance of imposing structure on a MIDI file. Using a functional MIDI programming approach, a function is applied on a sublist of the MIDI messages that makes up a song. In a very long lists of MIDI messages, it is not easy to identify the appropriate sublist.

Sections of a song (song verses, song lines, or other meaningful structures) can be delimited by a pair of markers. Individual markers are represented by MIDI Meta messages (of type 6 - marker events). As such, the markers survive a round trip between editing mode and creative mode (cf. Figure 1). The markers are most conveniently captured in creative mode at the MIDI instrument - recording the markers as notes in a distinguished channel. Different notes (C, D, E, ... H) denote markers at up to 7 levels. As a possible convention, each song verse may be marked by note D, and each song line may be marked by note C. These notes, in the distinguished channel, can subsequently be transformed to `Meta` messages by applying the library function `marker-channel` in a normalizing push operation on the

MIDI LAML stack.

It is possible to activate a tool which provides an overview of the markers in a given MIDI track - a *marker section browser*. A number of useful actions - such as playing, moving, deleting, and nesting of a marked region in certain function forms - can be activated from the marker browser. In addition, it is possible to embed an informal comment into a marker.

4.4 The Interactive Features

Although the main emphasis of the MIDI Programming environment is to support programmatic editing of MIDI LAML files, the environment also supports various interactive operations on normalized MIDI LAML source files. In many respects a MIDI LAML file resembles the detailed MIDI views in many existing MIDI tools and environments (sequencers). Therefore it is attractive to support some of the interactive operations known from such systems.

The most important category is the *MIDI playing* and *MIDI navigation operations*. It is possible to play (listen to) the whole song, a marked region of a song (delimited by markers, see Section 4.3), one or more bars of a song, the current selection of the song, all notes visible in an Emacs window, or the currently selected note. Convenient keyboard shortcuts - using arrow UP and arrow DOWN keys together with various shifting keys (Ctrl and Alt) - provide flexible access to the playing operations. Via use of function keys it is possible to control the playing tempo, and to select the the MIDI channels of interest. Channel selection affects both navigation and playing.

As a text editor, Emacs is not by itself able to play MIDI messages. Therefore the playing is delegated to an external MIDI player. With use of a command line player it is possible to emulate[3] rather smooth MIDI playing from Emacs. This includes a "moving cursor" which shows the note currently being played.

Attribute editing is supported via use of the arrow LEFT and arrow RIGHT keys (together with the Ctrl and Alt shifting keys). A given XML attribute - such as `note`, `velocity` or `absTime` - can be selected for editing. If the `note` attribute value is edited, the info attribute and the score comment (if present) is also affected. If the `absTime` attribute value is edited, the `NoteOn` form is automatically repositioned relative to sibling `NoteOn` forms. This maintain the MIDI LAML file normalization invariant (for absTime mode), as defined in Section 3.1. If the `deltaTime` attribute value is edited (in "compensating mode"), the delta times of neighbor MIDI events are modified as well, such that the rhythmic structure of music is maintained (like in absTime mode). All taken together, the attribute editing operations maintain a consistent view on the MIDI data.

[3]The emulation of smooth MIDI playing from Emacs involves two independent processes: The MIDI playing (done by the command line player) and the cusor moving (done by Emacs). On smaller sections of the music, and with moderate tempo, it is possible to keep these processes in sync with each other. It is also possible, interactively, to stop both processes at an arbitrary point in time (typically if some music-related error is revealed during the playing process).

In addition to the operations mentioned above, there are operations that interface external music programs, such as sheet music notation programs, and programs that catalyze concise note input (ABC).

4.5 Other features

The MIDI LAML programming environment includes a number of tools which are specific to a single MIDI instrument (keyboard), or to a family of instruments. This includes voices browsers, and browsers of very large collections of MIDI phrases (arpeggios and fragments of style[4] files). It is possible to activate the voices and MIDI phrases from the browser, on a selected instrument.

The voice and MIDI phrase browsers rely on an representations of the voices and the phrases that include various *meta data* (such as categorization and quantitative measures). Based on these meta data it is possible to zoom into and filter the collections, in search for some musical means of expression. On top of this, the textual nature of the environment makes it possible to do plain text searching, which typically is out of reach in more structured environments that rely on graphical user interfaces.

5. RELATED WORK

As mentioned in the beginning of the paper, there exists a very large body of MIDI software. In this section we will review some selected software which we are aware of, and which - for various reasons - are related to our approach (one reason being that the software is based on either Common Lisp og Scheme). We structure the discussion according to the programming paradigm being used.

5.1 Functional approaches.

Haskore [10], which we discussed already in Section 2.2, is a high-level music notation system, expressed directly in the functional programming language Haskell [9]. The starting point of the Haskore work was some observations about *structural and algorithmic weaknesses* in traditional music notation (common practice notation, sheet music notation). In Haskore, the remedy of these weaknesses is a set of Haskell datatypes and functions which can used to represent musical concepts at a high level of abstraction. The main concepts captured in the Haskore work are `Note`, `Line` (a sequence of notes), `Music` (a sequence of lines), as well as scaling, transposition, and sequential and parallel composition of `Music`. Representation of chords is also discussed in the Haskore paper. The high-level musical object, expressed as Haskell expressions, can be transformed to lower-level (MIDI-like) representation by means of a so-called *performance*.

The Haskore work represents a theoretical approach to Music notation, which emphasizes equivalence relations between pieces of Haskore music, algebraic properties, and mathematical proofs of musical properties. In contrast, MIDI LAML represents a practical approach, at a lower level of abstraction (similar to the performance level of Haskore), which emphasizes "real-life" transformation needs of music at the MIDI level. Due to the fact that both Scheme and

[4]In this context, a *style* is collection of MIDI phrases that, taken together, controls the automatic accompaniment of an 'arranger keyboard'.

Haskell are functional programming languages, it would be relatively easy to support high-level abstractions, similar of the Haskore ideas, on top of MIDI LAML. However, this has not yet been a focus area in our work with MIDI LAML.

Q-Midi [8, 7] is a MIDI system, based on the functional programming language Q and MidiShare [5] (a MIDI "operating system"). In Q-Midi the MIDI elements are provided as a set of datatypes. The Q-Midi datatypes are similar to the MIDI XML mirror functions, as provided in MIDI LAML. Q-Midi interfaces to MidiShare, and hereby real-time MIDI programming becomes possible via functional programming in Q. This provides for programming of MIDI sequencing programs, such as MIDI players and recorders, in Midi-Q. In MIDI LAML we emulate real-time MIDI capabilities with much simpler means.

5.2 Imperative approaches.

The Cakewalk Application Language (CAL) [21] is an imperative programming (scripting) language which is embedded in the commercial Calkewalk and Sonar products. These products represent Digital Audio Workstations (DAWs) and MIDI Sequencers. As a superficial similarity to our work, CAL is based on parenthesized prefix syntax - Lisp syntax - in the same way as Scheme. CAL and MIDI LAML are intended to solve the same kinds of problems, but in two different ways. CAL is based on commands and control structures, which mutate constituents of the MIDI contents, as represented in the MIDI sequencer program. In contrast, MIDI LAML is based on functions which return a modified copy of its input, in terms of an AST. The resulting AST can, for instance, be pushed onto the MIDI LAML stack as the starting point for additional processing. As a derived difference, CAL targets MIDI messages which are selected interactively in the surrounding MIDI sequencing environment. MIDI LAML functions do physically embed the target messages - as actual parameter expressions - in the activations of the functions.

5.3 Object-oriented approaches.

jMusic[5] [4] is a Java package for music composition. It supports both MIDI and synthesized audio. jMusic is an example of a system that represents an object-oriented approach to MIDI programming. Object-oriented programming is attractive in relation to MIDI programming because the structuring mechanism of object-oriented programming (specialization and aggregation) can used for a natural organization of MIDI concepts. The different types of MIDI events are represented by classes in jMusic, such as `NoteOn`, `NoteOff`, `PChange`, `CChange`, and `SysEx`. Common MIDI event properties are represented by an interface, `VoiceEvt`, as opposed to an organization in a class inheritance hierarchy. At a higher level of abstraction, the `Note` class captures a single note in a more elaborate and richer way than a pair of `NoteOn` and `NoteOff` MIDI messages. The `Rest` class is a natural sibling to the `Note` class. Sequences of notes are organized hierarchically in `Phrases`, `Parts`, and `Scores` container classes. On top of theses classes, jMusic supports a rich landscape of Java packages that facilitate the need of music tool makers.

The scope of jMusic is much broader than MIDI LAML. jMusic includes support of concepts at the MIDI level, similar to our work. In addition, it supports higher level organizations in the realm of Haskore, which we have described in Section 5.1. Beyond these aspects, jMusic supports audio programming as well as graphical concepts, which can be used a constituents of computer music tools.

The original version of Common Music [20] was an object-oriented music composition environment based on Common Lisp and CLOS. In Common Music it is possible to work on music at the Lisp level, at a command interpreter level, and at the level of a graphical user interface (on selected platforms). Common Music supports a variety of music structures, somehow similar to Haskore. In addition, Common Music is intended as a platform for algorithmic composition. In a more recent version[6] (version 3) Common Music has been oriented towards Scheme instead of Common Lisp and CLOS. As an addendum to Common Music, Common Music Notation[7] is a system that can produce non-trivial music scores based on specification written in Lisp list notation.

5.4 Visual approaches.

Musicians are seldom programmers, and therefore it may be hard for a musician to solve music-related problems with use of a traditional programming language. Visual programming uses graphical rather than textual means of expression. Consequently, it is interesting to support music-related problem solving with use of a visual programming language. In this section we will briefly review *OpenMusic* [1, 3] and related environments, which represent a visual approach to MIDI programming as well as audio programming.

The graphical part of OpenMusic draws on Mikael Laurson's system PatchWork [2]. PatchWork is rooted in the domain of music software, but part of it relates to visual programming in general. PWGL [13] represents the most recent development of the PatchWork environment. OpenMusic and PWGL have refined the basic ideas in PatchWork to the Common Lisp Object System (CLOS), and thus moved the foundation underneath the visual language to the object-oriented paradigm. Besides the graphical syntax of CLOS, OpenMusic and PWGL support a large variety of editors, which represents a library of music related components and applications (contributed by users of Open Music). In relation to MIDI programming, OpenMusic lends it self to MidiShare in a similar way as Q-Midi, see Section 5.1. As the name suggest, OpenMusic is basically situated in the open source community. However, the system relies to a large degree on commercial implementations of Common Lisp. In addition, some applications/editors in OpenMusic are not covered by open source licenses.

In relation to MIDI LAML, it is interesting to notice that the roots of OpenMusic and MIDI LAML belong in the functional programming paradigm. Both systems are based on dialects of Lisp. OpenMusic has appealed to real-world usage, most likely because of the graphical approach to programming, and due to the integration of existing non-trivial tools.

[5]http://jmusic.ci.qut.edu.au/

[6]http://commonmusic.sourceforge.net/

[7]https://ccrma.stanford.edu/software/cmn/cmn/cmn.html

6. CONCLUSIONS

In this paper we have discussed a representation of Standard MIDI files, which is both inspired by functional Lisp programming and by XML. The result is, on one hand, an attractive and well-known data representation which is easy to construct, validate, and transform due to the similarities with other XML document formats. On the other hand the result is a bulky textual notation, which is far away from common, high-level graphical music notation. Most important, we have found the representation is a flexible basis for high-level functional MIDI programming, and a realistic representation of *real music* (as opposed to more academic toy music) as created from MIDI files delivered by modern electronic instruments.

The MIDI LAML system is not a mainstream tool for MIDI work. It is a system which allows arbitrary changes of MIDI files, controlled by a functional program written in Scheme. Users of the MIDI LAML systems therefore need to be inclined to functional programming, and to master programming in the Scheme programming language. From a operational point of view, LAML MIDI files are handled in the Emacs text editor. As described in Section 4, the environment offers relatively broad support - including interactive listening/playing features. This calls for users who feel at home in the Emacs text editor. On top of this, the system relies on a number of separate programs (Cygwin, MIDI players, ABC software, in addition to GNU Emacs and MzScheme (PLT Scheme)) which must be configured relative to each other in the MIDI LAML system. Due to these observations and complications, the user-base of the MIDI LAML system is envisioned to be rather limited. At the time of finishing this paper, the author is the only user of the system.

LAML and MIDI LAML is available as free, GPL-licensed software which can be downloade via the MIDI LAML homepage [17].

7. REFERENCES

[1] Carlos Agon, Jean Bresson, and Gérard Assayag. Openmusic: Design and implementation aspects of a visual programming language. `http://recherche.ircam.fr/equipes/repmus/-bresson/docs/agon-els08.pdf`, 2008. Presented at the 1st European Lisp Symposium ELS'08, Bordeaux, France, 2008.

[2] Gérard Assayag, Camilo Rueda, Mikael Laurson, Carlos Agon, and Oliver Delerue. Computer-assisted composition at IRCAM: From patchwork to openmusic. *Computer Music Journal*, 23(3):59–72, 1999.

[3] Jean Bresson, Carlos Agon, and Gérard Assayag. Openmusic 5: A cross-platform release of the computer-assisted composition environment. In *Proc. 10th Brazilian Symposium on Computer Music, Belo Horizonte, Brazil*, 2005. http://articles.ircam.fr/textes/Bresson05b/.

[4] Andrew R. Brown. *Making music with Java.* lulu.com, 2009.

[5] D. Fober, Y. Orlarey, and S. Letz. Midishare joins the open source softwares. In ICMA, editor, *Proceedings of the International Computer Music Conference*, pages 311–313, 1999.

[6] M. Good. MusicXML in practice: Issues in translation and analysis. In *Proceedings of the First International Conference MAX 2002: Musical Application Using XML*, pages 47 – 54, September 2002. `http://www.recordare.com/good/max2002.html`.

[7] Albert Gräf. Q: A functional programming language for multimedia. In *LAC2005 Proceedings, 3rd International Linux Audio Conference*, pages 21–28. Zentrum für Kunst und Medientechnologie, Karlsruhe, Germany, April 2005.

[8] Albert Gräf. Q-midi: A midishare interface for the Q programming language. `http://q-lang.sourceforge.net/lac05/q-lac05.pdf`, March 2005.

[9] Paul Hudak and Joseph H. Fasel. A gentle introduction to Haskell. *ACM Sigplan Notices*, 27(5), May 1992.

[10] Paul Hudak, Tom Makucevich, Syam Gadde, and Bo Whong. Haskore music notation - an algebra of music. *Journal of Functional Programming*, 6:465–483, 1995.

[11] J. Hughes. Why functional programming matters. *Computer Journal*, 32(2):98–107, 1989.

[12] Richard Kelsey, William Clinger, and Jonathan Rees. Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, August 1998.

[13] Mikael Laurson, Mika Kuuskankare, and Vesa Norilo. An overview of PWGL, a visual programming environment for music. *Computer Music Journal*, 33(1):19–31, 2009.

[14] Kurt Nørmark. Web programming in Scheme with LAML. *Journal of Functional Programming*, 15(1):53–65, January 2005.

[15] Kurt Nørmark. Deriving a comprehensive document from a concise document - document engineering in scheme. In Danny Dubé, editor, *The 8th Workshop on Scheme and Function Programming.* Départment D'Informatique et de Génie Logiciel, Université Laval, Canada. Technical Report DIUL-RT-0701, September 2007.

[16] Kurt Nørmark. A graph library extension of SVG. In *Proceedings of SVG Open 2007, Tokyo, Japan*, September 2007.

[17] Kurt Nørmark. The MIDI LAML home page, 2010. `http://www.cs.aau.dk/~normark/midi-laml/`.

[18] R.M. Stallman. Emacs: The extensible, customizable, self-documenting display editor. In D.R. Barstow, H.E. Shrobe, and E. Sandewall, editors, *Interactive Programming Environments*, pages 300–325. McGraw-Hill, 1984.

[19] Guy L. Steele. *Common Lisp, the language, 2nd Edition.* Digital Press, 1990.

[20] Heirich Taube. An introduction to common music. *Computer Music Journal*, pages 29–34, 1997.

[21] Ton Valkenburgh. Cakewalk application language programming guide, May 2009. `http://members.ziggo.nl/t.valkenburgh/-indexmidi.html?/t.valkenburgh/CAL.html`.

[22] W3C. Extensible markup language (XML) 1.0 (fifth edition), November 2008.
`http://www.w3.org/TR/REC-xml`.