A Wrapper for MIDI files from an Object-Relational Mediator System

Semester thesis in informatics for the University of Zürich

Written by

Martin Jost
Zürich, Switzerland
Studentnumber 97-709-968

Made at the
Department of Information Science
at Uppsala University, Sweden

Prof. Tore Risch

1 Introduction

In the 80ties the relational database models prevailed in the practice. In many fields of application of the commercial and administrative world they found and find still their use. And this is totally entitled, because for these areas, the data model is optimally developed.

In the beginning of the 90ties the database world expanded into the area of multimedia applications and one determined quickly that the until now well known relational database models were not able to suffice the new demands to describe more complex objects. The User of so-called non-standard applications (CAD/CAM etc.) demanded a database model with which they also could store and manage the objects of their world.

Due to this fact, the search for a new database model started. This time the goal was the object oriented database model.

The goal of this project is to extend the Object-Oriented (OO) multi-database system Amos II with a wrapper implemented in Java, which can access MIDI files trough the net by giving the wrapper an URL of a midi file and store it in the database. To understand what is AMOS II, I will give a brief description about it.

AMOS II is a descendant of AMOS, which has its roots in a main memory version of Hewlett Packard's DBMS IRIS. The entire database is stored in main memory and is saved on disk only through explicit commands. AMOS II can be used as a single user database, a multi-user server, or as a collection of interacting AMOS II multi-database servers. AMOS II has a functional data model with a relationally complete object-oriented query language, AMOSQL. The AMOS II data manager is extensible so that new data types and operators can be added to AMOSQL, implemented in some external programming language (Java, C, or Lisp).
AMOS II is implemented in C and runs under Windows 95/98 and Windows NT 4.0. For more information about AMOS II, see [1].

To achieve the goal, the project was divided in the following 6 issues:

1. I had to acquire consolidated knowledge about the MIDI standard and its protocols. I will expatiate this issue in chapter 1.
2. The second task was to look after and select a development environment especially for Java classes, which can access MIDI files. To my advantage Sun's JDK 1.3 natively supports MIDI and sampled audio access. The Java Sound API will be discussed in Chapter 3.
3. One of the most important tasks was the investigation of an object oriented database scheme for the MIDI structure according to the Java Sound API.
4. In the next phase I designed primitive foreign functions which take the name of a MIDI file as a parameter, accessing them trough the web and then emitting a stream of AMOS II objects and attributes, according to the schema, to the mediator.
5. In chapter 5 I will show that useful queries can be specified over the schema and therefore the usefulness of a foreign function for MIDI.
6. Last but not least, the whole application had to be packaged and installed in a running AMOS II environment at the University of Uppsala.

2 What is midi?

The Musical Instrument Digital Interface (MIDI) standard defines a communication protocol for electronic music devices, such as electronic keyboard instruments and personal computers. MIDI data can be transmitted over special cables during a live performance, and can also be stored in a standard type of file for later playback or editing.

MIDI is both a hardware specification and a software specification. To understand MIDI's design, it helps to understand its history. MIDI was originally designed for passing musical events, such as key depressions, between electronic keyboard instruments such as synthesizers. Hardware devices known as sequencers stored sequences of notes. Those sequences could control a synthesizer, allowing musical performances to be recorded and subsequently played back. Later, hardware interfaces were developed that connected MIDI instruments to a computer's serial port, allowing sequencers to be implemented in software. More recently, computer sound cards have incorporated hardware for MIDI I/O and for synthesizing musical sound. Today, many users of MIDI deal only with sound cards, never connecting to external MIDI devices. CPUs have become fast enough that synthesizers, too, can be implemented in software. A sound card is needed only for audio I/O and, in some applications, for communicating with external MIDI devices.

The brief hardware portion of the MIDI specification prescribes the pin outs for MIDI cables and the jacks into which these cables are plugged. This portion need not concern us. Because devices that originally required hardware, such as sequencers and synthesizers, are now capable of being implemented in software, perhaps the only reason for most programmers to know anything about MIDI hardware devices is simply to understand the metaphors in MIDI. However, external MIDI hardware devices are still essential for some important music applications.

The software portion of the MIDI specification is extensive. This portion concerns the structure of MIDI data and how devices such as synthesizers should respond to that data. It is important to understand that MIDI data can be streamed or sequenced. This duality reflects two different parts of the Complete MIDI 1.0 Detailed Specification:

- MIDI 1.0
- Standard MIDI Files

I will explain what streaming and sequencing means by examining the purpose of each of these two parts of the MIDI specification.

2.1 Streaming Data in the MIDI Wire Protocol

The first of these two parts of MIDI specification describes what is known informally as "MIDI wire protocol". MIDI wire protocol, which is the original MIDI protocol, is based on the assumption that the MIDI data is being sent over a MIDI cable (the "wire"). The cable transmits digital data from one MIDI device to another. Each of the MIDI devices might be a musical instrument or a similar device, or it might be a general-purpose computer equipped with a MIDI-capable sound card or a MIDI-to-serial-port interface.

MIDI data, as defined by MIDI wire protocol, is organized into messages. The different kinds of message are distinguished by the first byte in the message, known as the status byte. (Status bytes are the only bytes that have the highest-order bit set to 1). The bytes that follow the

status byte in a message are known as data bytes. Certain MIDI messages, known as channel messages, have a status byte that contains four bits to specify the channel number. There are therefore 16 MIDI channels; devices that receive MIDI messages can be set to respond to channel messages on all or only one of these virtual channels. Often each MIDI channel (which shouldn't be confused with a channel of audio) is used to send the notes for a different instrument. As an example, two common channel messages are Note On and Note Off, which start a note sounding and then stop it, respectively. These two messages each take two data bytes: the first specifies the note's pitch and the second its "velocity" (how fast the key is depressed or released, assuming a keyboard instrument is playing the note).

MIDI wire protocol defines a streaming model for MIDI data. A central feature of this protocol is that the bytes of MIDI data are delivered in real time – in other words, they are streamed. The data itself contains no timing information; each event is processed as it's received, and it's assumed that it arrives at the correct time. That model is fine if the notes are being generated by a live musician, but it's insufficient if you want to store the notes for later playback, or if you want to compose them out of real time. This limitation is understandable when you realize that MIDI was originally designed for musical performance, as a way for a keyboard musician to control more than one synthesizer, back in the days before many musicians used computers. (The first version of the specification was released in 1984.)

2.2 Sequenced Data in Standard MIDI Files

The Standard MIDI Files part of the MIDI specification addresses the timing limitation in MIDI wire protocol. A standard MIDI file is a digital sequence that contains MIDI events. An event is simply a MIDI message, as defined in the MIDI wire protocol, but with an additional piece of information that specifies the event's timing. (There are also some events that don't correspond to MIDI wire protocol messages.) The additional timing information is a series of bytes that indicates when to perform the operation described by the message. In other words, a standard MIDI file specifies not just which notes to play, but exactly when to play each of them. It's a bit like a musical score.

The information in a standard MIDI file is referred to as a sequence. A standard MIDI file contains one or more tracks. Each track typically contains the notes that a single instrument would play if live musicians performed the music. A sequencer is a software or hardware device that can read a sequence and deliver the MIDI message contained in it at the right time. A sequencer is a bit like an orchestra conductor, it has the information for all the notes, including their timings, and it tells some other entity when to perform the notes.

2.3 The basic structure of MIDI

Finally I want to give a short set over the basic structure of MIDI on a rather low level of the MIDI specification.

A standard MIDI file consists of a header and one or more tracks.

```
<Standard-MIDI-File> := <Header> <Track> [<Track>]...
```

The first 4 bytes of the header are the identifier of a MIDI file, followed by some information bytes.

```
<Header> := 4D 54 68 64 00 00 00 06 00 xx yy yy zz zz
```

`xx := {00 | 01}`	Type of the MIDI-File: Type 0: File has only one track, Type 1: File has several tracks.
`yy yy`	Number of Tracks
`zz zz`	Timestamp: Number of Time elements per ¼ note; should be divided trough 24. Common values are: 00 78 (120), 00 F0 (240), 01 E0 (480), 03 C0 (960).

A track consists of a header and one or more events separated by a time difference.

```
<Track> := <Track-Header><Time-difference><Event>[<Time-
           difference><Event>]…
```

The track header has following information bytes:

```
<Track-Header> := 4D 54 72 6B xx xx xx xx
```

Whereby xx xx xx xx is the length of the track in bytes without the track header itself.

The time difference between the events is measured in timestamps. The first time difference counts from the beginning of the piece of music to the first event of the track and is coded in variable length.

An event can be either a channel event, a meta event or a sysex event and is the elementary command to the playback device.

```
<Event> := {<Channel-Event> | <Meta-Event> | <Sysex-Event>}
```

Let us have now a closer look on those elementary commands, since they hold the most important information in a MIDI file and thereby they are also necessary for the modelling of the object oriented database scheme.

2.3.1 Channel-Events

A channel event is a command to a certain MIDI channel of the playback device. The general syntax is as follows:

```
<Channel-Event> := [<Status-Byte>] <Data-Byte> [<Data-Byte>]
```

The status byte contains the type of the event and the channel number, on which the playback device executes the event.

```
<Status-Byte> := <Event-Type> <Channel-Number>
```

The first (leading) nibble of the status byte is the event type; it specifies the type of the command. Common values are 0x90 (Note On) and 0x80 (Note Off), see table 1 for details.

```
<Event-Type> := {8 | 9 | A | B | C | D | E}
```

The second nibble of the status byte declares the number of the MIDI channel which to perform the command on. The numbering starts with 0.

```
<Channel-Number> := {0 | … | F}
```

The data byte contains the parameter of the command. See Table 2 for explicit values and signification.

```
<Data-Byte> := {00 | … | 7F}
```

Setting the status byte only in the first event and omitting it in the following events can shorten a chain of channel events with the same status byte. This use is practically always made. The coding is explicit, since data bytes are smaller than 80 bytes and status bytes greater or equal 80 bytes.

MIDI message	HEX value	Dec value	Description
Active Sensing	0xFE	254	Status byte for Active Sensing
Channel Pressure	0xD0	208	Command value for Channel Pressure (Aftertouch) message
Continue	0xFB	251	Status byte for Continue message
Control Change	0xB0	176	Command value for Control Change message
End of Exclusive	0xF7	247	Status byte for End of System Exclusive message
MIDI Time Code	0xF1	241	Status byte for MIDI Time Quarter Frame message
Note Off	0x80	128	Command value for Note Off message
Note On	0x90	144	Command value for Note On message
Pitch Bend	0xE0	224	Command value for Pitch Bend message
Poly Pressure	0xA0	160	Command value for Polyphonic Key Pressure (Aftertouch) message
Program Change	0xC0	192	Command value for Program Change message
Song Position Pointer	0xF2	242	Status byte for Song Position Pointer message
Song Select	0xF3	243	Status byte for MIDI Song Select message
Start	0xFA	250	Status byte for Start message
Stop	0xFC	252	Status byte for Stop message
System Exclusive	0xF0	240	Status byte for System Exclusive message
System Reset	0xFF	255	Status byte for System Reset message
Timing Clock	0xF8	248	Status byte for Timing Clock message
Tune Request	0xF6	246	Status byte for Tune Request message

Table 1: Summary of MIDI Status Bytes

Type of command	1. Nibble of the Status Byte	1. Data Byte	2. Data Byte
Note On Turn on of a MIDI note. The note is played until a suitable command "Note Off" happens.	0x90	Tone pitch, for detailed information about the values see table 3. In percussion, the tone pitch is interpreted as a certain percussion instrument.	Velocity for the Note On event. Value 00 is illegal.
Note Off There are two possibilities.	0x90	Tone pitch	00
	0x80	Tone pitch	Velocity for the Note Off event. From most devices ignored.
Aftertouch The velocity of a note is changed.	0xA0	Tone pitch on which the command causes, if there is a note turned on.	Modified velocity
Control Change	0xB0	Number of the regulator. There are some predefined regulators in the MIDI specification, e.g. 00 = Choice of the sound bank 07 = Volume 0A = Stereo position	Controlled variable
Program Change Choice of the melodic instrument or drums.	0xC0	Number of the program. The assignment of instruments is determined in the MIDI specification. Enhanced standards use the regulator sound bank for sound variation.	None
Channel Pressure Changing of the velocity of all running notes in the channel.	0xD0	Modified velocity	None
Pitch Bend Changing the running notes tune.	0xE0	Pitch bend	

Table 2: Summary of most common MIDI Status & Data Bytes

Octave #	C	C#	D	D#	E	F	F#	G	G#	A	A#	B
0	0	1	2	3	4	5	6	7	8	9	10	11
1	12	13	14	15	16	17	18	19	20	21	22	23
2	24	25	26	27	28	29	30	31	32	33	34	35
3	36	37	38	39	40	41	42	43	44	45	46	47
4	48	49	50	51	52	53	54	55	56	57	58	59
5	60	61	62	63	64	65	66	67	68	69	70	71
6	72	73	74	75	76	77	78	79	80	81	82	83
7	84	85	86	87	88	89	90	91	92	93	94	95
8	96	97	98	99	100	101	102	103	104	105	106	107
9	108	109	110	111	112	113	114	115	116	117	118	119
10	120	121	122	123	124	125	126	127				

Table 3: Summary of MIDI Note Numbers for Different Octaves

2.3.2 Meta-Events

Meta events are not transmitted to the playback device. They control the playback tempo and provide additional information for better reading and understanding of a MIDI file. The general syntax is as follows:

```
<Meta-Event> := FF <Event-Type> <Length> [<Content>]
```

The type of the meta event is determined by the <Event-Type>. The <Length> parameter contains the following content in bytes. There is no content, if the length equals 0. Based on that syntax, meta events can be delimited in a data stream. A MIDI program doesn't have to know therefore the content of all meta events.

Selected meta events are represented in order of their meaning (the most important first).

Event-Type		Length	Content	Description
Track End	2F	00	None	Must stand at the end of each track. The event is represented including the time difference by "00 FF 2F 00"
Tempo	51	03	xx xx xx	Tempo in microseconds per quarter note.
Time Signature	58	04	xx yy zz 08	Beat in MIDI ticks per quarter note.
Key Signature	59	02	xx yy	xx > 0: key has xx crosses, xx < 0: key has –xx B's, xx = 0: C-major/a-minor; yy = 0: major, yy = 1: minor.
Track Name	03	Any	Any	Name of the track, usually description of the instrument. The name of the first track is interpreted often as song title.
Instrument	04	Any	Any	All sequencers do not indicate description of the instrument.
Text	01	Any	Any	Any comment
Song Text	05	Any	Any	Text module to sing with. Should stand directly after the relevant Note On event in the melody track. Does not become indicated by all sequencers.
User-defined	7F	Any	Any	Any

Table 4: Selected meta event types

2.3.3 Sysex-Events

These are system exclusive commands that are used by the MIDI standard. In a standard MIDI file, sysex events are represented as follows:

```
<Sysex-Event> := {F0 <Length> <Content> | F7 <Length> <Content>}
```

The <Length> parameter indicates the length of the following content in bytes.
Long command sequences are supposed to be transmitted in little portions with certain intervals. Therefore there are several syntactic variants, that all can be led back to both named variants:

Representation in the file	Significance	To transmit…
F0 <Length> <Commands> F7	Complete sysex event. The final signal F7 is included in the length.	F0 <Commands> F7
F0 <Length> <Commands>	The first portion of a separated event.	F0 <Commands>
F7 <Length> <Commands>	Sequel of a separated event.	<Commands>
F7 <Length> <Commands> F7	End of a separated event. The end signal F7 is included in the length.	<Commands> F7

Table 5: Syntactic variants of sysex events

3 The Java Sound API

The Java Sound API is a low-level API for effecting and controlling the input and output of sound media, including both audio and Musical Instrument Digital Interface (MIDI) data. The Java Sound API provides explicit control over the capabilities normally required for sound input and output, in a framework that promotes extensibility and flexibility.

The Java Sound API provides the lowest level of sound support on the Java platform. It provides application programs with a great amount of control over sound operations, and it is extensible. For example, the Java Sound API supplies mechanisms for installing, accessing and manipulating system resources such as audio mixers, MIDI synthesizers, other audio or MIDI devices, file readers and writers and sound format converters. The Java Sound API does not include sophisticated sound editors or graphical tools, but it provides capabilities upon which such programs can be built. It emphasizes low-level control beyond that commonly expected by the end user.

The Java Sound API includes support for both digital audio and MIDI data. These two major modules of functionality are provided in separate packages:

- javax.sound.sampled
  This package specifies interfaces for capture, mixing and playback of digital (sampled) audio.
- javax.sound.midi
  This package provides interfaces for MIDI synthesis, sequencing and event transport.

Two other packages permit service providers (as opposed to application developers) to create custom software components that extend the capabilities of an implementation of the Java Sound API:

- javax.sound.sampled.spi
- javax.sound.midi.spi

The rest of the chapter I will concentrate on the classes of the javax.sound.midi package, which are relevant for accessing and manipulating MIDI files.

3.1 The Java Sound API's Representation of MIDI Data

3.1.1 MIDI Messages

`MidiMessage` is an abstract class that represents a "raw" MIDI message. A "raw" MIDI message is usually a message defined by the MIDI wire protocol. It can also be one of the events defined by the Standard MIDI Files specification, but without the event's timing information. There are three categories of raw MIDI message, represented in the Java Sound API by these three respective `MidiMessage` subclasses:

- `ShortMessages` are the most common messages and have at most two data bytes following the status byte. The channel messages, such as Note On and Note Off, are all short messages, as are some other messages.
- `SysexMessages` contain system-exclusive MIDI messages. They may have many bytes, and generally contain manufacturer-specific instructions.

- `MetaMessages` occur in MIDI files, but not in MIDI wire protocol. Meta messages contain data, such as lyrics or tempo settings, that might be useful to sequencers but that are usually meaningless for synthesizers.

3.1.2 MIDI Events

As we've seen, standard MIDI files contain events that are wrappers for "raw" MIDI messages along with timing information. An instance of the Java Sound API's `MidiEvent` class represents an event such as might be stored in a standard MIDI file.

The API for `MidiEvent` includes methods to set and get the event's timing value. There's also a method to retrieve its embedded raw MIDI message, which is an instance of a subclass of `MidiMessage`. (The embedded raw MIDI message can be set only when constructing the `MidiEvent`.)

3.1.3 Sequences and Tracks

As mentioned earlier, a standard MIDI file stores events that are arranged into tracks. Usually the file represents one musical composition, and usually each track represents a part such as might have been played by a single instrumentalist. Each note that the instrumentalist plays is represented by at least two events: a Note On that starts the note, and a Note Off that ends it. The track may also contain events that don't correspond to notes, such as meta events (which were mentioned above).

The Java Sound API organizes MIDI data in a three-part hierarchy:

- `Sequence`
- `Track`
- `MidiEvent`

A `Track` is a collection of `MidiEvents`, and a `Sequence` is a collection of `Tracks`. This hierarchy reflects the files, tracks, and events of the Standard MIDI files specification. (Note: this is a hierarchy in terms of containment and ownership; it's not a class hierarchy in terms of inheritance. Each of these three classes inherits directly from java.lang.Object.)

`Sequences` can be read from MIDI files, or created from scratch and edited by adding `Tracks` to the `Sequence` (or removing them). Similarly, `MidiEvents` can be added to or removed from the tracks in the sequence.

3.2 Use of the Java Sound API

In this chapter I want to show some basic functions of the Java Sound API relating to my foreign function. It should show how the classes and interfaces of the Java Sound API are used to stream a MIDI file through the net and accessing its tracks, events, and so on. The following code fragments are taken of my foreign function and can be read more detailed in Appendix A.

In order to load a MIDI file into a sequencer and finally into AMOS II, we have to deliver an URL of an existing MIDI file to the foreign function. We take the URL parameter of the foreign function from Amos toploop to create an URL object:

```
String s = tpl.getStringElem(0);
    try {
      url = new URL(s);
    } catch (MalformedURLException e) { System.out.println(e);
}
```

The base module in the Java Sound API's messaging system is `MidiDevice` (a Java language interface). `MidiDevice` include sequencers (which record, play, load, and edit sequences of time-stamped MIDI messages), synthesizers (which generate sounds when triggered by MIDI messages), and MIDI input and output ports, through which data comes from and goes to external MIDI devices. In our case we need to load a sequencer to access the MIDI sequence of our URL object. We also try to load a synthesizer, so we can extend in future our foreign function to play back a MIDI sequence.

```
Sequencer sequencer;
try {
     // Try to load the default Sequencer
     sequencer = MidiSystem.getSequencer();
        if (sequencer instanceof Synthesizer) {
          synthesizer = (Synthesizer)sequencer;
          channels = synthesizer.getChannels();
        }
    } catch (Exception ex) {ex.printStackTrace(); return;}
```

Now we create a sequence, we invoke `MidiSystem`'s overloaded method `getSequence`. The method is able to get the sequence from an InputStream, a File or an URL like in our case. The method returns a `Sequence` object, which we can access now:

```
Sequence currentSound = MidiSystem.getSequence(url);
// Get some information about the MIDI Sequence
duration = currentSound.getMicrosecondLength() / 1000000.0;
tickLength = currentSound.getTickLength();
resolution = currentSound.getResolution();
divisionType = currentSound.getDivisionType();
```

To get a reverence to the available `Tracks` in the `Sequence` we create a `Track` object and invoke `Sequence.getTracks`:

```
Track tracks = currentSound.getTracks();
```

Once the sequence contains tracks, we can access the contents of the tracks by invoking methods of the `Track` class. The `MidiEvents` contained in the `Track` are stored as a `java.util.Vector` in the `Track` object, and `Track` provides a set of methods for accessing, adding and removing the events in the list. The methods `add` and `remove` are fairly self-explanatory, adding or removing a specified `MidiEvent` from a `Track`. A `get` method is provided, which takes an index into `Track`'s event list and returns the

`MidiEvent` stored there. In addition, there are size and tick methods, which respectively return the number of `MidiEvents` in the track, and the track's duration, expressed as a total number of `Ticks`.

```
// Loop all MidiEvents in a Track object
for (int i=0; i < tracks.length; i++) {
     // Get number of events in a track
     events = tracks[i].size();
     ticks = tracks[i].ticks();
     // Create the MidiEvent object in a Track object
     midiEvent = new MidiEvent[events];
          // loop the MidiEvent object
          for (int j = 0; j < events; j++) {
               // obtain the event at the specified index
               midiEvent[j] = tracks[i].get(j);
               timestamp = midiEvent[j].getTick();
               midiMessage = midiEvent[j].getMessage();
               // Check if the MidiEvent is a MetaEvent, a
               // ChannelEvent or a SysexEvent
                    if (midiMessage instanceof MetaMessage) {
                    // Handle the MetaMessage object and create
                    // the necessary objects and functions in
                    // AMOS II
                    //…
                    }
                    if (midiMessage instanceof ShortMessage) {
                    // Handle the ShortMessage object and
                    // create the necessary objects and
                    // functions in AMOS II
                    //…
                    }
                    if (midiMessage instanceof SysexMessage) {
                    // Handle the SysexMessage object and
                    // create the necessary objects and
                    // functions in AMOS II
                    //…
                    }
          }
}
```

For more detailed information about the classes, interfaces and methods of the Java Sound API, see the Java 2 Documentation [8].

4 Database schema

After we made ourselves now a picture over the Java Sound API, we take a look at the object oriented data base scheme. One of the most important targets of this project consisted of developing an object oriented database model in order to be able to store MIDI files in AMOS II. There were certain basic conditions to consider. First of all the database model is as far as possible supposed to be object oriented, i.e. to deploy the most important concepts of object orientation, which AMOS II places to use. Secondly the MIDI specification should be illustrated as complete and meaningful as possible. And thirdly I could refer to the classes of the Java Sound API at the development of the database model. In the following I want to describe briefly my database model as seen in fig. 1. For more detailed information about the implementation of the database model in AMOS II, see the database creation scripts in Appendix B.

The database model consists of eight types, whereof three are inherited and three serve as lookup types for some translations of data. The program takes a loaded MIDI file as a sequence, in which all tracks and events are contained as we have seen in the previous chapters. The sequence is decomposed in accordance with the MIDI specification into single elements and then stored as objects into AMOS II. The type MIDI represents the core MIDI object, it contains following attributes:

MIDI

URL: String of the URL, which we downloaded the file of
FileName: String of the name of the file
Length: Integer of the size of the file in KB
Duration: Real of the Length of the file in seconds
NumberOfTracks: Integer of the number of existing tracks in the MIDI file
TickLength: Integer of the duration of the MIDI sequence, expressed in MIDI ticks
Resolution: Integer of the timing resolution for the MIDI sequence. That is the number of ticks per beat or per frame.
DivisionType: Real of the timing division type for this MIDI sequence

As we know already, the MIDI sequence consists of at least one track. These Track objects are designed under the type Track and are connected over the relation tracks with its MIDI object. Since there can be several tracks in a MIDI sequence, the relation is implemented as a set of objects. The type Track has the attributes:

Track

NumberOfEvents: Integer of the number of events in this track
Ticks: Integer of the length of the track expressed in MIDI ticks

The single events in the track are created as objects of type MidiEvent in AMOS II. Each single MidiEvent is connected with its Track object by the relation midievents. Since there can be several MidiEvents per Track object, the relation is implemented as a vector of MidiEvent.

MidiEvent

TimeStamp: Integer obtaining the time stamp for the event in MIDI ticks
Length: Integer of the number of bytes in the MIDI event, including the status byte and any data bytes
StatusByte: Integer obtaining the status byte for the MIDI event

MidiMessage: Byte array containing the MIDI event data. The first byte is the status byte for the event; subsequent bytes up to the length of the event are data bytes for this event. In our database scheme the bytes are transformed into integers and stored as a vector of integers.

The created MidiEvent object can either be a channel event, a meta event or a sysex event (system exclusive). This is now the point where our database scheme gets object oriented. All this different events are implemented as a type under the type MidiEvent and therefore are directly inherited from MidiEvent. Each event (channel, meta and sysex) has the same attributes, TimeStamp, StatusByte and MidiMessage, which are inherited of the MidiEvent type. The types ChannelEvent, MetaEvent and SysexEvent are connected through an is-a relation with the super type. Besides the inherited attributes, each event type has its own attributes.

ChannelEvent

Channel: Integer obtaining the MIDI channel associated with this event
Command: Integer obtaining the MIDI command associated with this event
Data1: Integer obtaining the first data byte in the message
Data2: Integer obtaining the second data byte in the message
CommandAsString: String representing the command translated to a meaningful String
NoteEvent: derived function to translate the channel event command into a meaningful string. Information is retrieved from type NoteEvents.
Note: String representing the note of a Note On or a Note Off channel event.
Octave: Integer obtaining the octave of a Note On or a Note Off channel event

MetaEvent

MetaType: Integer obtaining the type of the meta event
MetaDataInt: Obtains a copy of the data for the meta message. The returned array of bytes does not include the status byte or the message length data. The length of the data for the meta message is the length of the array. Note that the length of the entire message includes the status byte and the meta message type byte, and therefore may be longer than the returned array. In our database scheme the bytes are transformed into integers and stored as a vector of integers if the data is representing an integer value of the MIDI file.
MetaDataString: Same as MetaDataInt, except it contains a string. If the data of the meta event represents a string like lyrics, text or instrument description, the data is stored in MetaDataString.
MetaTypeAsString: String representing the meta type translated to a meaningful String

SysexEvent

SysexData: Obtains a copy of the data for the system exclusive message. The returned array of bytes does not include the status byte. The array of bytes is transformed into integers and stored as a vector of integers in our database scheme.

These are all the types and functions that are needed to model the object oriented database scheme for the MIDI 1.0 specification. Because a lot of data is stored as simple integers we extend the database model with three ‘lookup’ types in order to translate the integers into meaningful data. In fact it is now much easier for the user to search for a specific string instead of an integer or to understand how the data is represented in a MIDI file. The three types are ChannelCommands, NoteEvents and MetaCommands. The ChannelCommands type is connected with the channelEvent type through the derived function commandAsString. It

takes the integer of the command of a channel event and translates it into a string. The NoteEvents type, which is connected through the derived noteEvents function, takes the data of a Note On or a Note Off event and translate it into the particular Note. Finally, the function MetaTypeAsString takes the integer of the metaType and translates it through the MetaCommands type into a string.

ChannelCommands

Command: Integer obtaining the channel commands.
commandString: String representing the translated integer value of a channel command.

NoteEvents

noteInt: Integer obtaining the integer value of a Note On or Note Off event.
noteString: String representing the note of a Note On or Note Off event as a string.
Octave: Integer obtaining the octave of a Note On or Note Off event.

MetaCommands

MetaTypeInt: Integer obtaining the meta type of a meta event.
MetaTypeString: String representing the meta type translated into a string.

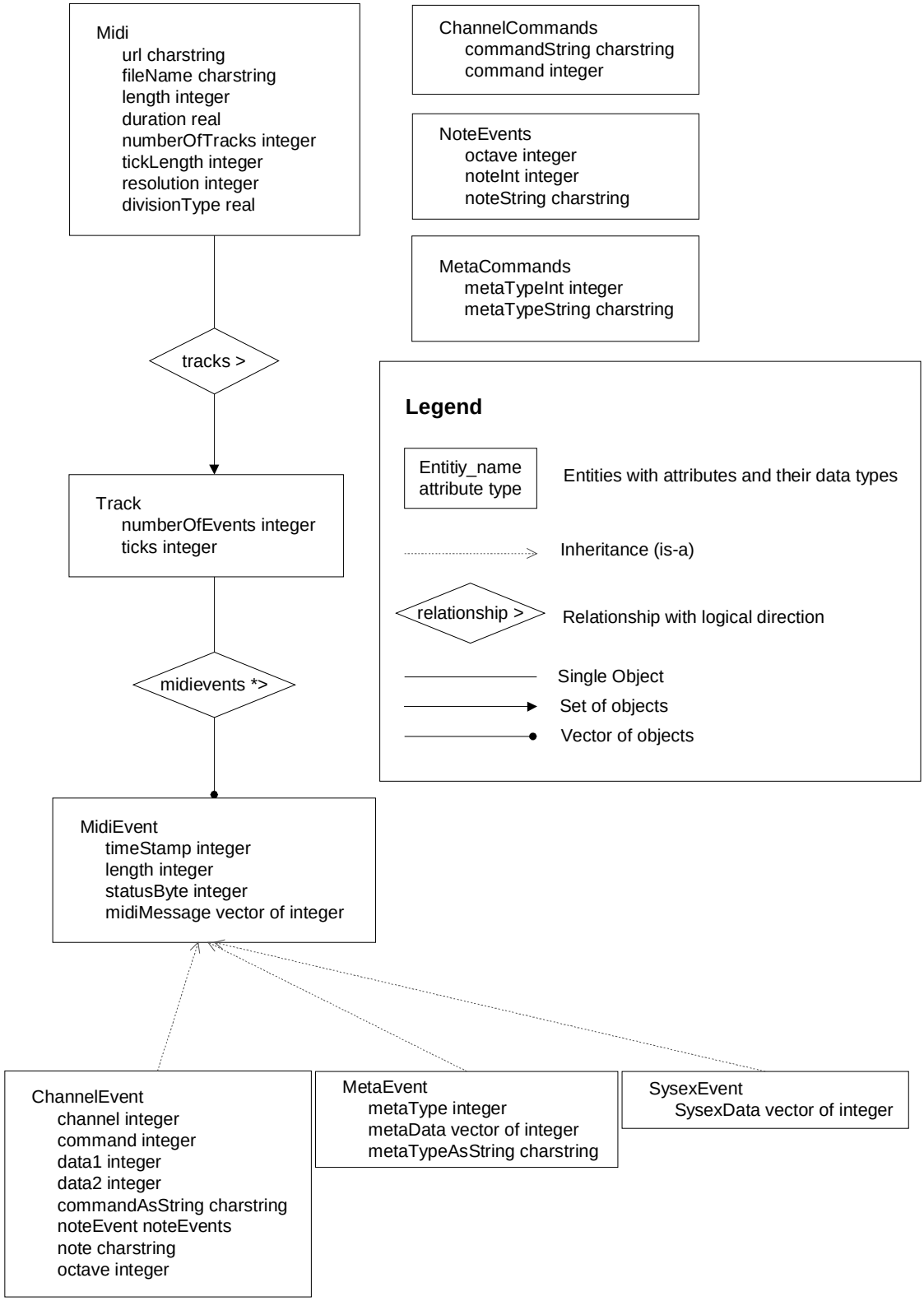

Fig.1: The object oriented database scheme for Midi

5 Examples

Well, let us see now what are the benefits of this MIDI wrapper for the object-oriented database AMOS II. In this chapter I want to show some reasonable queries to approximate the usefulness of this wrapper. We will load some MIDI files in our AMOS II database through the web and make some queries to retrieve selected information.

First of all we need to set up the environment properly and create the database. Refer to Appendix C to see the installing instructions.

Once the environment is running, we load some MIDI files into AMOS II:

```
JavaAMOS 1> loadMidi("http://files.midifarm.com/midifiles/
general_midi/sinatra_frank/myway4.mid");

JavaAMOS 2> loadMidi("http://files.midifarm.com/midifiles/
general_midi/roxette/roxette-sleepinginmycar.mid");
```

After we loaded as many MIDI files as we want into AMOS II we save the database by the command:

```
save  'midi.dmp';
```

Now we can make some queries with AMOS' II query language AMOSQL. Similarly to SQL, AMOSQL is a combination of DDL (Data Definition Language) and a DML (Data Manipulation Language). The query part of the language is similar to SQL.

The following command answers the question 'Which song last longer than 2 minutes and has two or more Tracks'.

```
JavaAMOS 1> select distinct m
           from midi m
           where duration(m) > 120.00 and tracks(m) >= 2;
```

Or we can simply search for a MIDI object of a certain interpret:

```
JavaAMOS 2> select m
           from midi m
           where like(filename(m),"*sinatra*");
```

We can also define functions to retrieve or manipulate data in AMOS II. A function defined in terms of other functions is called a derived function. Especially if we often need to look for specific information in the database, we define derived functions. There are already some derived functions defined in our database scheme. For example to find out which track or which midi object a midi event belongs to, we have defined the following derived functions in the database scheme:

```
create function track(midievent e)-> track t as
       select t
       from integer i
       where
```

```
        midievents(t)[i] = e;

create function midi(midievent m)-> midi as
       select midi(track(m));
```

Now we can use these derived functions to get for example all MIDI objects that play a guitar as instrument:

```
JavaAMOS 3> select distinct midi(m)
              from metaevent m
              where like(metaDataString(m), "*guitar*");
```

To answer the question 'Which songs are written in C-major?' we have to perform several commands. As you may remember, key signature data is stored as a meta event in our database scheme. The key signature has the meta type value 89 that is translated into the string "Key Signature". The data itself is represented as a vector of two integers. To find now a song written in c-major, we have to look for a vector with two zeros as integers. For more details see table 4.

First we create a derived function to look for the specific vector of integers that represents a c-major song:

```
JavaAMOS 4> create function major(metaevent m)->metaevent as
              select m from metaevent m
              where metadataint(m)[0]=0
              and metadataint(m)[1]=0
              and metatypeasstring(m) = "Key Signature";
```

The following command will now retrieve all the MIDI songs written in c-major:

```
JavaAMOS 5> select major(m) from metaevent m;
```

Analogical we can look for all MIDI songs written in a-minor:

```
JavaAMOS 6> create function minor(metaevent m)->metaevent as
              select m from metaevent m
              where metadataint(m)[0]=0
              and metadataint(m)[1]=1
              and metatypeasstring(m) = "Key Signature";

JavaAMOS 7> select minor(m) from metaevent m;
```

There are many other valuable queries that can be executed. Last but not least I want to show a query, which can be improved later on to find patterns of note events in a MIDI object. We want to retrieve all Note On events on a specific channel of a certain MIDI object in order of their appearance. First we have to create a derived function to compare two channel events ordered by their timestamps:

```
JavaAMOS 8> create function orderByTimestamp(channelevent ca,
channelevent cb) -> boolean as select true where timestamp(ca)
< timestamp(cb);
```

The following command will retrieve all Note On events in channel four of the MIDI song ‘My way’ by Frank Sinatra. The `sort` function will order the results regarding our derived function `orderByTimestamp`.

```
JavaAMOS 9> sort((select ce from channelevent ce where
command(ce) = 144 and channel(ce) = 4 and
filename(ce)=”/midifiles/general_midi/sinatra_frank/myway4.mid
”), “orderByTimestamp");
```

6 Summary

Today we find databases in many kind of use to store every kind of data we can imagine. The goal of this project was now to access any given midi file through a database wrapper and store the data of the MIDI file in an object-oriented database called AMOS II. This database wrapper is implemented in Java and runs as a foreign function in AMOS II. The function takes the name of a MIDI file as a parameter, accesses it trough the web and then emits a stream of AMOS II objects and attributes, according to the developed schema, to the mediator.
To understand how this works I want to give a brief overview over the MIDI specification and the object-oriented database scheme. How does the MIDI structure look like? Each MIDI stream can be divided in its single tracks; those tracks in turn consist of different sequential events. There are three types of different events, the channel events, the meta events and the system exclusive events. Most important though are the channel events, since they include information about how to play each tone. Meta events are not transmitted to the playback device; they control the playback tempo and provide additional information for better reading and understanding of a MIDI file.
The object-oriented database scheme has been developed to carry all the information of a MIDI file, which we can access through the Java wrapper. There are tree base types, Midi, Track and MidiEvent and three types, which are inherited of the base type MidiEvent. Those types are ChannelEvent, MidiEvent and SysexEvent. To store all the data in the database, the wrapper creates the necessary database objects out of the MIDI stream according to our scheme. The MIDI file exists now in our database as a numerous and various set of objects.
What are the benefits of such a MIDI wrapper? There are plenty of reasonable queries to approximate the usefulness of such a wrapper. You may be interested in all the songs of a specific interpret, or you are looking for a song written in c-major and played on a piano. Since every single note event is stored as an object, you could also search for a pattern of notes to retrieve the requested song.
The future work for this project will surely be to implement a foreign function that plays back a MIDI track through AMOS II. This foreign function should select a MIDI object and load it into a sequencer to play back the song. Also many other fields of applications can be implemented, such as editing tracks or events. For example adding new tracks to a MIDI object or removing them, changing the instrument on a channel or adding lyrics or text to the MIDI object. Pattern matching functions can also be invented to search for a pattern of note events. That might be very useful if you remember a piece of the melody of a song, but have forgotten the name or interpret.

7 References

[1] Tore Risch, Vanja Josifovski, Timour Katchaounov: *AMOS II Concepts*, June 23, 2000
http://www.dis.uu.se/~udbl/amos/doc/amos_concepts.html

[2] Staffan Flodin, Vanja Josifovski, Timour Katchaounov, Tore Risch, Martin Sköld, and Magnus Werner: *AMOS II User's Manual*, AMOS II Beta Release 3, June 23, 2000
http://www.dis.uu.se/~udbl/amos/doc/amos_users_guide.html

[3] Tore Risch: *Introduction to AMOSQL*, Department of Information Science, Uppsala University, Sweden, April 23 2001
http://www.dis.uu.se/~udbl/engdb/amosql.pdf

[4] Gustav Fahl and Tore Risch: *AMOS II Introduction*, Uppsala Database Laboratory, Department of Information Science, Uppsala University, Sweden, October 1, 1999

[5] Daniel Elin and Tore Risch: *AMOS II Java Interfaces*, Uppsala Database Laboratory, Department of Information Science, Uppsala University, Sweden, August 25, 2000

[6] Joe M.: *MIDI Specification 1.0*
http://www.ibiblio.org/emusic-l/info-docs-FAQs/MIDI-doc/

[7] Rainer Jahn: Spezifikation MIDI-Datei, December 5, 1999
http://www.zbs-ilmenau.de/~rainer/midispez/

[8] *Java 2 Platform SE v1.3.1:Package javax.sound.midi*,
http://java.sun.com/j2se/1.3/docs/api/javax/sound/midi/package-summary.html