Uppsala Student Thesis in

Computer Science 310

2007-05-27

Searching contents of wrapped MP3 files from an Object-Relational Mediator System

Thomas Schreiter

Information Technology
Computing Science Department
Uppsala University
Box 337
S-751 05 Uppsala
Sweden

Supervisor: Erik Zeitler
Examiner: Tore Risch

Abstract

Amos II is a mediator system developed at the database department of Uppsala University. It provides the Query Language AmosQL, which is an object oriented expansion of SQL. ISO-MPEG Audio Layer-3 files (also known as MP3) contain music. One wants to treat an MP3 file as a source of data and analyze its content. To query an MP3 file via Amos II, one has to wrap the MP3 files to get access to them. The basic approach to this integration is to send the data from the MP3 file to Amos II in a streamed fashion. This paper describes the work to build such a wrapper.

Contents

1 Introduction and Motivation

There are some technologies how to store music in a digital way. This paper deals with the most popular version of music files nowadays: the ISO-MPEG Audio Layer-3 files, better known as the abbreviation MP3.

These files are divided into two parts. The first part contains meta-data about the song. Information about the author, the length of the song, the sampling rate, the version number of the used encoding technique and much more is stored there. The second part of the MP3 file contains the actual music. To analyze the actual content of a music file the binary coded music has to be cracked out of the files and be decoded into a stream.

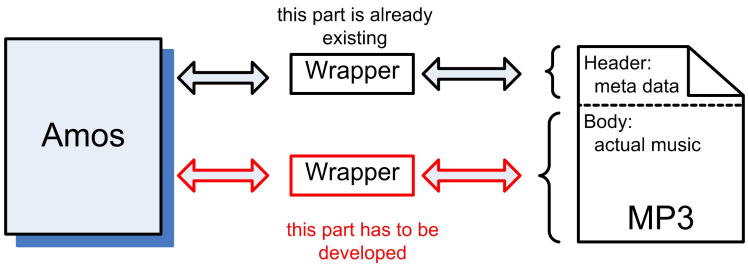

Figure 1: Content of this paper

The Amos II mediator system enables the integration of different types of data from different sources of data. Usually, the data sources are a priori not compatible with the Amos II system. To get access to the data source we have to implement an adapter between the data source and Amos. The adapter will translate the access functions of the data source into functions of the Amos system. Such an adapter is also called a wrapper.

There is already a wrapper existing [15]. This wrappers enables the querying of the *meta data* of an MP3 files. These data are stored in the file separately from the actual encoded music. These are information about the author of the song, the genre of the song, the used encoding technique and more. This paper covers the development of a wrapper between Amos II and the *music* or content of MP3 files (Fig. 1). The content has to be accessed and wrapped with functions to provide the information to Amos II and its querying language AmosQL. It will be a demonstration of AmosQL's flexibility.

So, we will connect the two objects "Amos II" and "MP3 file" by their data at a low level. This technique of connecting two or more systems to one working unity by their data structures is called Information-oriented Application Integration [12].

This paper's structure is as follows: Chapter 2 specifies the task of this project, including the wrapper and the filter functions. Furthermore the the Amos II system and the structure of music files will be analyzed. The design of the wrapper and the filters are described in Chapter 3. These techniques will be implemented, as Chapter 4 shows. One will also find the description of usage, the installation and a demonstration of this integrated system. An summary and outlook is given in chapter 5.

2 Analysis of the given systems and Specification of the Wrapper and the Filter

2.1 Specification of the new system

The specification of the system is brief and informal and divided into two parts:

1. "Enable a mechanism to read out the data of an MP3 file. The focus is on the music itself, not on the meta-data."

   This part will provide the decoded music to the Amos environment. So we have to wrap the MP3 files. This part of the project will be referred to as the wrapper part.

2. "Enable a mechanism to perform an analysis on the MP3 music data to extract features. In this part use the language features of AmosQL."

   This part avails the results of the first part. We will develop interesting analyzing functions and implement them with the language features of AmosQL. This part of the new system will be referred to as the filter part.

This specification is written generally. Thus, it provides flexibility to adapt if there are serious problems.

In the next chapter we will analyze the system Amos and we will take a closer look at MP3 technology. After that we can outline the parts of the new system more clearly. Then we design the new system in chapter 3.

2.2 Analysis of Amos II

This chapter introduces the most important concepts and constructions of the Amos II system. At the homepage of the database laboratory of the Uppsala university are detailed tutorials located which are worth reading [11, 8]. In [7] Amos is described as:

> Amos II (Active Mediator Object System) is a distributed mediator system that uses a functional data model and has a relationally complete functional query language, AmosQL. [...]. Functional multi-database queries and views can be defined where external data sources of different kinds are translated through Amos II and reconciled through its functional mediation primitives. [...] The Amos II data manager and query processor are extensible so that new application oriented data types and operators can be added to AmosQL, implemented in some external programming language (Java, C, or Lisp). The extensibility allows wrapping data representations specialized for different application areas in mediator peers. The functional data model provides very powerful query and data integration primitives which require advanced query optimization.

The core application of AmosQL is the usage of query statements. AmosQL is an extension to SQL, so it enables the SQL typical `select` statements as well as `create` and the `insert` statements.

Amos II's goal is to connect different data sources to one working database. To access and use the stored data, Amos enables a typing system which allows to define own types. Basic types like integer and real numbers are implemented as well as vectors and bags. New types can be developed be inheriting the features of an existing type or be defining a completely new one.

Amos supports functions. Three different concepts will be introduced:

- *Stored functions* do not compute the return value in a mathematical sense, but it looks up the result in its own database. The result is stored in the database.
- *Derived functions* are composite functions. They use the SQL part with its `select` operations or call other functions to calculate their result.
- *Foreign functions* are implemented in an foreign language. AmosQL is specialized to database queries and is not adequate to compute difficult numerical algorithms. Such algorithms can be implemented in a foreign programming language. A foreign function is basically a call to the interface of the foreign programming language. There exist interfaces to Java, C and Lisp.

2.3 The technology of MP3

In this part we will analyze the structure of an MP3 file.

MP3 is standardized by the MPEG Committee. This group is engaged in creating standards for the compression of audio and video signals. It defines the syntax of audio and video format and the operations of decoders.

The meta data of an MP3 file

As mentioned before, an MP3 file is divided into two parts – meta data of the song and the actual music. The meta data about song are stored in a table at the header of the file – or sometime in the trailer, depending on the encoding technology. It contains information about

- the used encoding technique and version
- the number of channels (mono/stereo)
- the sampling frequency and the nominal bit rate
- the length in bytes and in frames
- the frame rate, the frame size of the first frame and other very specific information about the encoding
- the copyright of the song
- the author, the album and the genre of the song

This first part of an MP3 file is, as mentioned before, already wrapped for the Amos II system [15].

The actual music in an MP3 file

The second and major part of an MP3 file contains the actual, encrypted music. We will take a short look at the techniques of encoding music to an MP3 file [10].

Music, or sound in general, is the vibration of air. These vibrations are perceived by the human ear and transformed in the human brain to a sensory perception. The human being cannot recognize all types of sounds. We only can hear sounds of frequencies between 16 Hz and 20 kHz. Whereby we are more sensitive in perception in some frequencies than in others. For example, we are very sensitive at hearing quiet sounds in the range between 2 kHz and 5 kHz.

Additionally, in an almost silent environment we can hear the sound of our own breathing. But, in contrast, in a loud room we do not hear our breathings, because our senses are distracted by the noise. These two "features" of the human sense of hearing are called non-linear audio threshold and masking effect, respectively. These are in great use of reducing the amount of data while encoding music. This is called perceptional coding.

During the compression, some minor and unimportant details of the music are omitted. The song details the human being is not capable of hearing are suppressed by the perceptional coding. So, some information get lost, but the result is not changed in a great manner. Comparing the original song with an encoded and re-decoded song shows no difference to the human ear. The songs *seem* to sound the same.

There are some more techniques used in the encryption algorithm. Since the two streams of the left and the right stereo do not differ that much, they are coded bundled by the Joint Stereo coding. At the end of the compression the calculated symbols are finally encoded with a Huffman code. This is a very fast algorithm that saves up to 20 % of space.

The encoding takes a lot of time. But the initial bitrate of 768 kBit/sec are reduced to 128 kBit/sec or even less.

An MP3 file is structured as follows: It consists of a header containing the meta-data and a body containing the actual music.

The body consists of frames which have the following features:

- Each frame has a 32 bit header containing information about the synchronization and encoding of the frame
- Each frame represents audio data of a duration of 26 ms, that are 38 frames per second
- Each frame contains exactly 1152 samples
- The sampling frequency is $F_s = 44\,kHz$

The way of encoding and decoding MP3 files is an asynchronous one. The encoding takes a lot more time than the decoding. But the latter is much faster. It is possible in real time, obviously. A lot of operations have to be performed. Figure 2 depicts the necessary operations. The MPEG committee standardized these operations, so there are lots of decoders built and available.

The input to the decoder is an MP3 file. The decoder skips the meta data of the MP3 file and scans the frames of the body. The information of each frame header are used to synchronize the decoding, enable the correct Huffman decoding scheme, enable the correct bitrate and the other important decoding parameters. The decoding procedure contains the decoding of the Huffman symbols, a cosine transform, the stereo decoding, Alias Reduction, and some other functions. The final stage transfers the signal into the (discrete) time domain representing a (discrete) sound wave. So, at no point during the decoding do the data represent the sound in the frequency domain.

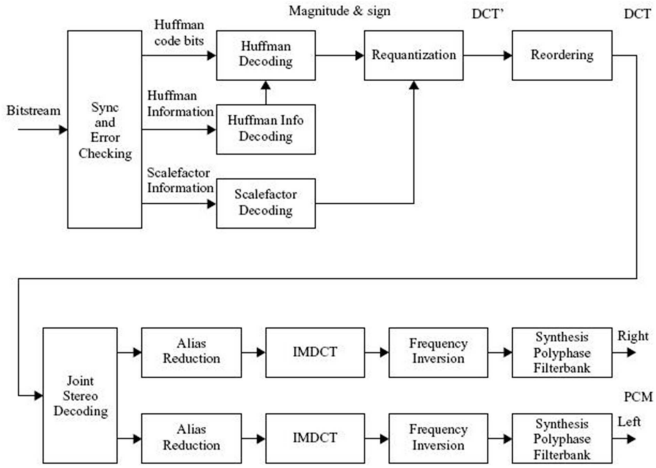

Figure 2: MP3 decoder structure [9]

The conclusion: The encoded data in the MP3 file are a priori not useful for any analysis of the music. While decoding the file, data are transformed to different stages, but no one of them represents the information in a useful way. The result of the decoding is a bit stream representing the time domain of the sound. Every 16 bit are one PCM sample (PCM stands for Pulse Code Modulation). A positive value is a positive value of the amplitude, a negative value a negative amplitude at a certain point in time. So, the whole output stream is a digital waveform, quantized in time and amplitude.

3 Design of the Wrapper and the Filter

3.1 General Issues

We have three major parts to regard.

1. Accessing the MP3 files, decoding them and delivering the result to Amos II.
2. Analyzing the result by filter functions in Amos II.
3. Storing the extracted features in the Amos II database.

The first big part of decoding an MP3 file cannot be done by the language capabilities of AmosQL. So this part will be implemented by a foreign function. The design of the decoding part will be discussed in section 3.2.

The filter functions will deal only with feature extractions of low level. AmosQL provides simple functions like addition, multiplication, minimum, vector operations and so forth. Additionally, queries known from the SQL `select`-statement serve as a good basis to design the filter functions with the concept of derived functions. Chapter section 3.3 explains the design of the filter.

These attained result will be stored in the Amos system. AmosQL and its concept of stored functions are the perfect instrument so create a

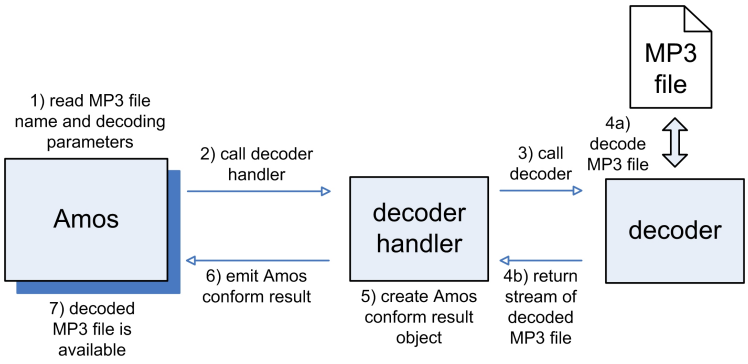

Figure 3: Design of the Wrapper

database of the analyzed songs.

3.2 Design of the Wrapper

Amos does not provide built-in functionalities to access and decode MP3 files. So, we will use a foreign program to achieve the desired function of retrieving decoded PCM samples of MP3 files. For that reason we have to develop an own program to access and handle the actual MP3 decoder. It acts as a handle of the decoder.

The basic way of accessing the MP3 files is as follows (Figure 3):

1. in AmosQL: get the file name of the MP3 file and the decoding parameters
2. in AmosQL: call the foreign decoder handler
3. in decoder handler: open the MP3 file and call the actual decoder
4. the decoder returns the result in a stream to the decoder handler
5. in decoder handler: format this result to an object that Amos can understand
6. in decoder handler: emit this object to AmosQL, end after MP3 file is decoded
7. in AmosQL: receive the decoded PCM samples

The decoder

The main and central point of this wrapper design is to find an MP3 decoder. In the Internet one can find hundreds of decoders. But only a fistful of them are satisfactory enough to be worth to be used. The following list describes some of them, including the reasons why I have chosen or rejected them, respectively:

- The madlib library [6] provides a good and general basis to decode MP3 files. It has been build at the year 2000 and being improved until 2004. A lot of other programs use its functionalities. Unfortunately, the documentation is very small and does not give the necessary information for this thesis. But a least a little demonstration program is attached to show an instance of usage.

So I had to read the source code by my own. The programming language used is C, which promises a fast execution time. The code is very generally written, the use of function pointers provide a useful flexibility. But on the other hand, some very C special features and qualities might provide a fast decoding algorithm, but it is very hard to read for a new developer. To show three examples: assignment in an if-condition, more than one return command in a function, function calls in combination with boolean operators in an if-condition. And to top it off, the goto-commands make it virtually impossible to understand the code within few hours.

Conclusion: It might be a great program, but the lack of a comprehensive documentation is a killing factor. We have only a small project and need only the basic functionality of decoding an MP3 file. So it is not worth to get that deep into the library.

- Audiere [1] is a library to decode music files. A documentation is available. But, again, it is a small documentation. Anyway, at the moment Audiere is not supporting MP3 file due to the license of the decoder algorithms. Maybe they will include it again in the near future. So Audiere is not a choice now.
- The decoder called mpg123 [3] has very good test results. Unfortunately, the latest version has a huge security bug and it is not further developed.
- Java provides an MP3 decoder within its distribution. The usage of audio streams let the code look short and clear. We will use the code of the already existing MP3 meta data wrapper [15], that contains also a little routine to playback MP3 files by using audio streams. So that seams to be the best way to decode MP3 files into PCM Streams. The runtime might be slower than the decoders written in C, but the short development time is the evident reason to use Java.

So, our choice will be to use Java as decoder. For that reason the handler of the decoder will be implemented in Java, too. The both parts "decoder" and "decoder handler" will be implemented in Java in one source code file.

The decoder handler

From Amos to the Decoder: After a foreign call by Amos, the decoding information are extracted and the decoder will be constructed. The decoder is basically an object of the class `AudioInputStream`, which is part of the distribution of Java 1.5.

From Decoder to Amos: The decoder returns a stream of PCM samples in a byte-oriented way. The samples values have to be calculated. They are then bundled to vectors of PCM samples. After that they are send to Amos.

3.3 Design of the Filter Functions

As discussed previously, the decoder emits vectors of PCM samples of an MP3 file to Amos II. We have to design analysis functions to extract features of the MP3 file:

- The maximum amplitude of a song is highest value of the music signal in the time domain. It is simply an iteration over all PCM samples and returning the highest value. The function name will be `AnaSongMaxAmplitude()`.
- The energy spectrum of a song is an analysis of the frequency spectrum of the song. It is a value describing the dispersion of the energy value of each frequency in the song. This is a nontrivial computation and we will discuss it in more detail soon. The name of the function will be `AnaSongEnergySpecSmall()`.
- The frequency with the highest energy is that frequency in the song, that is stronger than all others. This is an analysis of the energy spectrum and returns a frequency/energy pair describing the strongest frequency in Hz and its energy. The energy itself has no absolute unity, because the PCM samples have no unity. The name of the function will be `AnaSongHiFreq()`.

The Energy Spectrum

The signal analysis theory serves the theoretical background for extracting information of a signal. The whole process of calculating the energy spectrum is shown in Figure 4.

As described before, music is the vibration of air. More scientifically spoken, it is a signal in the continuous time domain, described by the series $\{y(t)\}$. To use digital computers, this signal is been transformed into signal in the discrete time domain $\{y_n\}$ by sampling with a sampling frequency F_s. So, two consecutive PCM signals y_i and y_{i+1} are amplitudes in the time domain with the pitch of $1/F_s$. The decoder returns such a PCM series $\{y_n\}$.

At first we will transform the time domain signal $\{y_n\}$ into the frequency domain signal $\{Y_k\}$ by the Fast Fourier Transform:

$$Y_k = \sum_{n=0}^{N-1} y_n e^{-\frac{2\pi i}{N} kn}, \quad k = 0, 1, \dots N-1$$

where e is the Euler number, i is the imaginary unit and N a power of 2. Note, that the argument k has a domain between 0 and $N-1$. To calculate the frequency spectrum with a frequency parameter F in Hertz (Hz) we simply replace the argument k with

$$k = \frac{F \cdot N}{F_s}$$

This relationship is valid in all equations in this paper. In the following we use the equations with the parameter k. An example how to use the relationship: The usual sampling rate of an MP3 file is $F_s = 44\,kHz$, so the the frequency spectrum shows the density between $-22\,kHz$ and $+22\,kHz$ with a pitch of $\frac{44\,kHz}{N}$.

Note also, that the input to the FFT is a real valued signal. The output of the Fourier Transform is therefore even, meaning $Y_k = Y_{(N-1)-k}$. So we can truncate the vector $\{Y_k\}$ to its first half part.

The next step is the calculation of the intensity of each frequency. This is expressed with the power spectrum Φ_k:

$$\Phi_k = \left| \sum_{n=0}^{N-1} y_n e^{-\frac{2\pi i}{N} kn} \right|^2 = Y_k Y_k^*$$

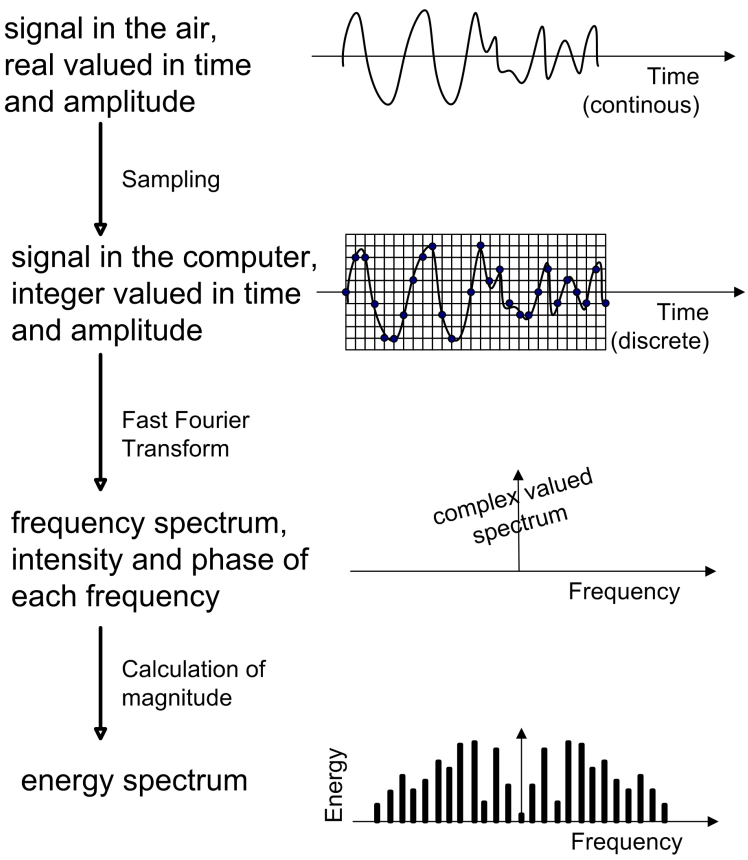

Figure 4: Calculation of the Energy Spectrum

This is the magnitude of each element of the vector.

Execution of `MP3SongEnergySpecSmall()`: the decoder output will be divided into P vectors of size N. We obtain a series $\{\{y_n\}_p\}$. The energy spectrum Φ_k of each vector is computed ($\{\Phi_k\}^p$) and afterward compressed to vector with only few elements ($\{\tilde{\Phi}_k\}^p$); each element representing a frequency band. Finally, the vectors are added

$$\tilde{\Phi}_k^{Song} = \sum_{p=0}^{P-1} \tilde{\Phi}_k^p$$

representing the frequency spectrum of the song in a compressed form.

4 Implementation and Usage of the Wrapper and the Filter

4.1 General Issues

We want to decode MP3 files and run analysis over them. The result will be short vectors or simple numbers, so it will be only a negligible small amount of data compared to the size of one MP3 file. These results will be stored in the Amos system by stored functions. We will refer a stored function as to a record.

One record contains the filename of the MP3 file, as well as the decoding parameters and the result parameters. Let's analyze all fields of one record:

- The `filename`: denotes the name of the file of the desired MP3 file and the relative path starting from the working directory.
- Decoding parameter `numFreqs`: denotes the number of resulting frequencies in the result of the frequency analysis. Higher values show a better resolution in the frequency domain, lower values show describe the spectrum in a more compact way. Usually a relative small number between 2^2 and 2^5 is interesting, to keep the result concise. A power of 2 is also useful, because a lot of Fast Fourier Transforms will be performed, which provides the best runtime with inputs of the size of powers of 2.
- Decoding parameter `channel`: denotes the number of the channel. If one wants to decode and analyze only one channel.
- Decoding parameter `winSize`: The Java decoder doesn't emit PCM sample one after another. But they are bundled to a bunch of samples and emitted together at once. The order of the samples is not changed. The reason to do this is to reduce the number of foreign function calls to reduce the execution time. `winSize` denotes the size of the PCM sample vector, that the Java decoder emits to the Amos system. Basing on this vectors, further analysis will be performed. Usually a Fast Fourier Transform is performed, so a power of 2 between 2^8 and 2^{16} is useful.
- Decoding parameter `normFactor`: To perform a Fast Fourier Transform the PCM samples are converted from integer to complex. Furthermore the values are normalized to values in range of -1 and 1. At the moment the PCM sample size is fixed to 16 bit, so the only possible value is 2^{15}.

- Result parameter `energySpectrum`: denotes the energy spectrum of the MP3 file. It is a real vector with a size of `numFreqs`.
- Result parameter `highestFreq`: denotes the frequency with the highest energy in the song.
- Result parameter `highestEnergy`: denotes the energy of that frequency.
- Result parameter `maximumAmplitude`: denotes the value of the highest amplitude in the time domain.

4.2 Implementation of the Wrapper

As described earlier, the foreign functions of the wrapper are written in Java. The most important functionalities are getting access to an MP3 file, extracting the PCM samples and sending them to Amos in a streamed fashion.

The Java Code

The functionality of accessing an MP3 file is coded in Java. This code is called externally by AmosQL query statement.

We will now take a look at the Java code and discuss some of the methods. The source code of Java is structured in four classes:

- The class `MusicFile2` contains methods to access and decode a music file. It encapsulates a filename and functions to create file streams.
- The class `ContentFunctions` acts as an interface to Amos and contains the functions that are accessed by the AmosQL query language. These functions contain only the necessary functionality to process the call by Amos.
- The class `MP3content` implements the actual functionality. For the sake of clarity the actual functionality is detached from the call processing. The emission of the resulting PCM samples is also implemented in this function.
- The `Main` class contains basically only the call to the Amos top loop to start Amos.

The Heart of the Extraction Algorithm

The most important functionality of the Java part is the extraction of the PCM samples of an MP3 file. The input of the algorithm is the filename of the desired MP3 file. The necessary routines for the basic handling of an MP3 file is coded in the class `MusicFile2`.

After a foreign call by the AmosQL language, an instance of this class is being created. The constructor of `MusicFile2` is called with a String parameter symbolizing the filename of the desired MP3 file.

The music stream of the MP3 file is accessed by two instances of the type `AudioInputStream`. One handling the coded MP3 file (before), the other one handling the decoded PCM samples of the MP3 file (after). The method `getAis()` opens the first stream of the file. It contains the byte of the body of the MP3 file. This stream is identified in the code with `ais`.

This stream is decoded into the second stream. The method `getDecs()` opens the Java built-in decoder to decode `ais`. The decoding parameters are set in the class `AudioFormat`. We want to decode our MP3 file into

PCM samples, so we set the format for the decoded stream in the method `getDescFormat()`: The encoding scheme is `AudioFormat.Encoding.PCM_SIGNED`, the output sample size is set to 16 bit. The parameter sample rate, number of channels and size of an PCM frame (this is not an MP3 frame!) depend on the MP3 file itself. Therefore these parameters are derived from the `ais` stream format. The resulting stream is identified in the code with `decs`.

The decoded stream `decs` contains a byte stream of the decoded PCM samples. Unfortunately, the samples are not accessible directly. Firstly, the stream operations allow only reading of bytes. It is not possible to read one PCM sample without any further computation. Secondly, the samples of both channels are merged, meaning the order in the stream is: first PCM sample of the left channel, first PCM sample of the right channel, second PCM sample the left channel, second PCM sample of the right channel and so forth.

Therefore we have to calculate the PCM sample values by hand and separate the channels by ourselves. The method `DecodeFileToAmos()` contains a loop labeled with an accordant comment. It works as follows:

- Reading bytes of `decs` and storing them in a buffer
- As long there are bytes in the buffer, reading two bytes and computing the PCM sample value
- Storing the PCM sample value in the result buffer of one channel
- Switching to the result buffer of the other channel

The Interface to Amos

The concept of foreign functions enables the implementation of algorithms in a foreign language. The Java function `AmosCallFileDecode()` of the class `AmosForeignFunction` is the entrance to the Java decoder handler. It is published in AmosQL via the command:

```
create function DecStreamVec(charstring filename,
    integer smplWinSize)
    -> <Vector of integer smpl, integer chn>
  as foreign
  'JAVA:ContentFunctions/AmosCallFileDecode';
```

The result is sent to Amos via the `emit` command:

```
cxt.emit(tpl);
```

where `cxt` is an `Context` object and `tpl` and `Tuple` object.

In Java the function parameters and results are stored in an instance of the class `Tuple`.

4.3 Implementation of the Filter Functions

After the MP3 file is decoded, the PCM samples are processed by the AmosQL environment.

The filter functions are implemented in Amos by the concept of derived functions.

It follows the most important wrapper functions; their names in AmosQL, a description of the meaning, and a short description of their implementation. Table 1 summarizes them with a measuring of the run time.

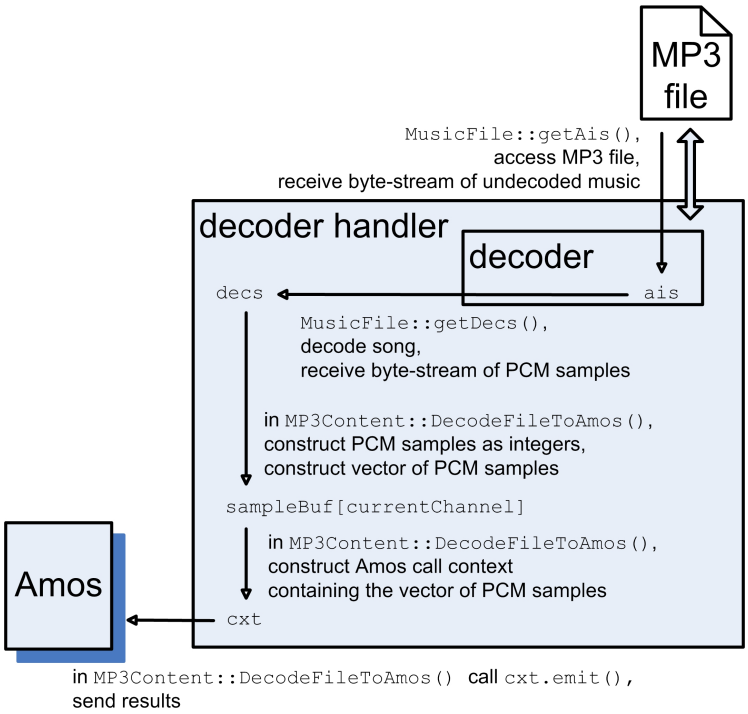

Figure 5: Implementaion of the Wrapper

- The function `Decode()` returns the PCM samples from an MP3 file defined in an `MP3Song` object. The PCM samples are bundled in vectors of integers of 16 bit size, meaning a sample value is between -32768 and 32767.

  This is the first and basic function, since it provides the raw material which the other filter function are performed on. It is also the only filter function that calls the foreign Java decoding code.

- The function `AnaTimeDomain()` returns the PCM samples from an MP3 file. The PCM samples are bundled in vectors, too. But a sample is a not an integer as in the previous function, but a real value normalized to a number between -1.0 and 1.0.

  This function calls the function `Decode()`.

- The function `AnaFreqDomain()` returns the normalized frequency spectrum. They are bundled in vectors of complex numbers.

  This function calls the functions `Decode()` and the Fast Fourier Transform `FFT()`. Note that the function `AnaTimeDomain()` is bypassed. The `FFT()` returns values with a precision of 0.1. Since the time domain is normalized to number numbers less than 1, the `FFT()` performed directly on the normalized time domain would provided very coarse and hence worthless results. For that reason, `FFT()` takes the unnormalized time domain as input, after that, that result is normalized.

- The function `AnaEnergySpec()` returns the energy spectrum. They are bundled in vectors of complex numbers.

Name	Result	Calls Function...	Run Time
`Decode()`	PCM samples, vector of 16 bit integer	foreign Java	4.2 sec
`AnaTimeDomain()`	PCM samples, vector of real	`Decode()`	17.3 sec
`AnaFreqDomain()`	frequency domain, vector of complex	`Decode()` and `FFT()`	77 sec
`AnaEnergySpec()`	energy spectrum for every sample window, vector of real	`AnaFreqDomain()`	136 sec
`AnaEnergySpecSmall()`	compressed energy spec for every sample window, vector of real	`AnaEnergySpec()`	150 sec
`AnaSongEnergySpecSmall()`	compressed energy spec for whole song, vector of real	`AnaEnergySpecSmall()`	155 sec
`AnaSongMaxAmpliude()`	maximum amplitude for whole song, real	`AnaTimeDomain()`	12.0 sec

Table 1: Filter functions in AmosQL

This function calls the function `AnaFreqDomain()` and calculates the magnitude of each single element of the frequency spectrum. The energy spectrum represents the intensity of each frequency.

- The function `AnaEnergySpecSmall()` calculates the energy spectrum, too. However the result is compressed to a vector of real of small size. The exact size is specified in the parameter `MP3Song.numFreqs`. This representation of the energy spectrum is easier to read for a human.

  This function calls the function `AnaEnergySpec()`.

- The function `AnaSongEnergySpecSmall()` returns the compressed energy spectrum for the whole song. This is a feature describing the song with a few numbers. Like the previous function it returns a vector of real.

  This function calls the function `AnaEnergySpecSmall()` and accumulates all vectors.

- The function `AnaSongMaxAmplitude()` returns the maximal amplitude occurring in the song. It a real value between 0 and 32768.

  This function calls the function `AnaTimeDomain()`.

The functions are summarized in Table 1. The column "Run Time" shows the measured runtime of a test MP3 file. It is a song of 37 seconds length. The decoding parameters are chosen as follows: `smplWinSize` = 8192, `numFreqs` = 16, `channel` = 0, `normfactor` = 32768. The song can be found in the main directory of the project under the `filename` = `''Helan.mp3''`. This MP3 file contains the Swedish song "Helan går".

The User Functions

To update the Database, the user calls functions that execute the calculations and updates the database. The names of these start with `MP3Song...` ("MP3Song" to indicate an analysis of the song including an update of the `MP3Song` instance in the database). The function argument of each function is an `MP3Song` instance.

Each one of these function is a composition of a call of one the `Ana...()` function and a `set` operation to modify the database. For every result attribute of the `MP3Song` type exists a user function:

- `MP3SongEnergySpectrum(song)` calculates the compressed energy spectrum of a song. It sets the attribute `song.energyspectrum`. It calls the function `AnaSongEnergySpecSmall()`.
- `MP3SongMaximumAmplitude(song)` calculates the highest amplitude of a song. It sets the attribute `song.maximumAmplitude`. It calls the function `AnaSongMaxAmplitude()`.
- `MP3SongHighestFrequency(song)` calculates the pair of the highest frequency and the energy of the highest frequency. It sets the attribute `song.highestFrequency`. It calls the function `AnaSongHighestFreq()`.

4.4 Installation

This subsystem is an "Add-On" to the existing Amos II system. So the prerequisite is the Amos system with all its files. An introduction to the system and its installation can be found in [8]. If one uses the CVS version of Amos II, this subsystem should be included in the directory /wrappers/MP3/MP3content.

The files fall in these categories:

- The *Java file* contains the foreign functions needed for the decoding of the MP3 files.
- The *AmosQL files* contain the data types and functions that are used in the Amos environment. The big part of these files realize the filter functions that operate on the decoded song.
- The *MP3 files* are the music files, obviously. These are three songs that are used for test issues. The results of "Helan.mp3" are shown in this paper. The other songs are "Weißes Rauschen" by the band "Die Toten Hosen" and "Got the Life" by "Korn". It can be used for own queries.
- Most important: The **.cmd files* to install the system by compiling the resources and generating the image. Call the file in this order:
  1. "setup.cmd" to setup environment variables
  2. "compile.cmd" to compile the Java file
  3. "mkdump.cmp" to generate the .dmp file
  4. "run.cmd" to finally execute the system
- The *music.dmp file* contains the image of the system. The foreign functions, the data types, the filter functions and the results of one test file are included. The Amos system is loaded with this image.

4.5 Demo and Usage of the System

After installing the system it is ready to use. Start Amos with the generated image file:

```
% javaamos mp3content.dmp
```

Alternatively, you can also use the command `run.cmd`.

Suppose we want to analyze the energy spectrum of an interesting MP3 file, let us say the file `Helan.mp3`. The steps we have to perform are roughly:

- Create a new `MP3Song` instance
- Set the analysis parameters of that instance
- Call the analysis function

We create a new `MP3Song` by typing:

```
% JavaAMOS 1> createMP3Song('Helan.mp3');
```

Note that there is no blank between `create` and `MP3Song`! Of course, we could also use the AmosQL `create` statement. But the function above already sets the filename attribute and the normFactor attribute.

Since we want to compute the energy spectrum, we have to choose the resolution of the result. Meaning how many "bars" the final energy spectrum has to contain. Let us say we want to have 16:

```
% JavaAMOS 2> set numfreqs(s) = 16
    from mp3song s where filename(s) = "Helan.mp3";
```

We also have to set some decoding parameters. These will not affect the format of the result, but play a role in the decoding and analyzing procedure. Lets say, we want to analyse channel number zero and the decoder shall send the PCM samples in bunches of 2^{13} to Amos:

```
% JavaAMOS 3> set channel(s) = 0
    from mp3song s where filename(s) = "Helan.mp3";
% JavaAMOS 4> set smplWinSize(s) = 8192
    from mp3song s where filename(s) = "Helan.mp3";
```

Finally, we have to decide, how much of the song we want to analyze. The default setting is to analyze the whole song (value -1). If we are interested only in the first 10 seconds, we set `numSamples` to 14 (1 represents 1.5 second for one channel, therefore in the stereo song: $\frac{10\,s}{2 \cdot 1.5\,s} = 13.3$:

```
% JavaAMOS 5> set numSamples(s) = 14
    from mp3song s where filename(s) = "Helan.mp3";
```

Now the configuration is complete and we can run the analysis function:

```
% JavaAMOS 6> for each MP3Song s
    where filename(s) = 'Helan.mp3'
    MP3SongEnergySpectrum(s);
```

The analyzer will run and compute the result. It will take some time, enough to have a coffee break. The result is stored in the attribute `energySpectrum`. To get the result we query:

```
% JavaAMOS 7> select energySpectrum(s)
    from mp3song s where filename(s) = 'Helan.mp3';
```

Of course, the configuration and analysis is also possible in two statements:

```
% JacaAMOS 1> create MP3Song( fileName, numFreqs, channel,
    smplWinSize, numSamples, normFactor ) instances
    :HelanFile ( 'Helan.mp3', 16, 0, 8192, 14, 32768.0);
% JavaAMOS 2> MP3SongEnergySpectrum(:HelanFile);
```

We can look at the result by

```
% JavaAMOS 3> EnergySpectrum(:HelanFile);
{13755317.2169852,427219.498719408,13740.4757344478,...}
```

The result shows 16 numbers. They represent the 16 "bars" of the energy spectrum. Each spectrum is $2756Hz$ wide ($\frac{44kHz}{16} = 2725Hz$). So the frequencies between 0 and $2756Hz$ have an energy of $1.3 \cdot 10^7$. The frequencies between $2756Hz$ and $5512Hz$ have an energy of $4.2 \cdot 10^6$ and so forth. Remember that this energy has no unity, since the PCM samples have no unity.

5 Summary and Outlook

Topic of this thesis was to build a connection between the mediator system Amos and the file format MP3, specializing in the music content of this file type. The goal was to put queries to the MP3 files containing content features like: "What is the highest amplitude in the song?" or "Name the top frequency that occurs in the song!"

For that goal we had at first to design and implement a system that accesses the files and returns the content of the file in a format that is easy to read and to compute. In our case we opened the MP3 file via the Java environment and decoded the MP3 format to PCM sample values in the (discrete) time domain, which basically represents the sound as a wave. These samples are emitted to the Amos system in bundles, the bundle size is chosen randomly, but usually one chooses a power of 2, because one wants to transfer the time domain signal to a signal in the frequency domain via the Fast Fourier Transform. The functions programmed in Java are published to Amos via the concept of foreign functions. A query in Amos calls a function of our Java code via a defined interface.

Second, we can perform filter functions on these PCM samples. These can be simple queries like finding the highest value of all samples ("What is the highest amplitude?"). This can be achieved in AmosQL with its language capabilities. More difficult operations like the Fast Fourier Transform need to be done by another call to a foreign implementation. In the case of the FFT, it was implemented by another developer of UDBL some months before. This example shows the flexibility and reusability of foreign functions.

5.1 Known issues

The implementation of this project does not contain flexibility in the sample size of the PCM sample values. So, more precise analysis (finer

than 16 bit) has to be done another way. I found no way to expand it to 24 bit. The documentation of Java does not go that deep into the data type of `AudioInputStream` that one can deeply understand the techniques of this data type and its relatives. There might be other data types from other service providers on the market, but I did not find anything in an appropriate time.

After the decoding of the MP3 stream the decoded stream contains the PCM samples in an channel-merged and a fragmented way. The calculation of the actual sample values is a bit extensive and looks like we want to break a butterfly on the wheel. But the built-in Java data types like `byte` and `integer` do not allow operations on a bit level. So operations that could easily work with bit shifting, bit masking and bitwise OR-operations have to be expressed with integer arithmetics. Maybe the Java virtual machine and its compilers and interpreters can perform these operation in a fast runtime.

A very big problem is the run time. Table 1 on page 16 shows some typical functions one wants to use to extract features of a file. The decoding itself does not take that long. It is only 12% decoding time compared to the actual playing time of the song. Some filter functions take an incredible long time, for example the calculation of the energy spectrum. Up to five times longer than the song actually plays.

But what is the reason for that lack of speed? The "default slow" Java Virtual Machine is not the problem, obviously. The Java Virtual Machine is no longer accessed after the MP3 file is decoded to the PCM samples by the function `MP3Decode()` or one of its relatives. So there are at least three other possible reasons.

1. The coding of the functions is badly chosen. In many cases the sample values (or frequency values, energy values,...) are bundled in vectors. To access and change a sample value, every single vector element has to be accessed separately. A code example of a function, that computes the magnitude of each element of a vector of complex:

   ```
   create function magnitudeVec(vector of complex cvec)
       -> vector of real rvec
     as
     select vectorof(
       select magnitude(cvec[i])
       from integer i
       )
   ;
   ```

   The double usage of the `select` statement slows the computation. An alternative would be to break up vectors immediately after decoding by `MP3Decode()` and treat the PCM sample values as a Bag. But usually one needs the computation of a FFT, which has as input parameter a vector.

2. The implementation of the Fast Fourier Transform has some potential left. A test with input vector size of 2^{14} might lead to page trashing, hence to a very long execution time or even to a crash. Unfortunately, at least a sample vector size of 2^{15} is desirable. Reason: The maximal frequency that is encoded in an MP3 file is around 16 kHz. So, to recognize these high frequencies one needs at least 2^{15} samples. Even better is a higher amount.

3. The implementation of the Amos system itself is slow. The operations on the vectors are slow, as discussed before. Maybe also the handling of large streams (Bags) can be enhanced. Additionally, some query optimization might be possible.

All these points need a deeper understanding of Amos and the FFT which cannot be provided in this thesis.

It would be interesting to see the same specification ("decode an MP3 file to its PCM samples") implemented in a programming language that is "naturally fast". There are many decoders on the market that are implemented in C. They are probably faster. But the big problem with existing programmes is to integrate them into the own existing systems. The biggest problem in our case was the missing documentation. So the easiest way in this project was to write an own decoder with the big help of Java. It might be a much longer run time, but it is also development time that matters.

5.2 Beyond...

A lot more filter functions are imaginable. As long as the filter functions realize the extraction of a sharply defined feature of the song, they are realizable in AmosQL directly or a as foreign functions implemented in Java or C.

However, a complexer analysis of the music might lead into very hard computations. The basic question is then: "How do you characterize music from a objective point of view?"

The initial point of this analysis is only a stream of PCM samples. To compute an answer to a question like: "To which genre does this song belong?" is indeed a very difficult task. Any queries that go further than statements like"Compute the energy spectrum" cannot be tackled in a paper like this. It would be by far too complicated. But luckily, in the last years evolved an own scientific field of music analysis. Lots of people participate in the research field of music analysis and recognition. Every year there are conferences held in several subdisciplines of this new area of science. The appendix of this paper shows a little impression of some music analysis programs.

But be careful! This field of science is a new one and the programs are more or less in a experimental stage. They are not perfect, but some of them are available on the market.

A Notes on Feature Extraction of Music

This chapter explains some work in the field of Features Extraction. The reason why this chapter is included in this paper is simple. We thought about including a high level feature extractor to our system and use its results to classify music. So it took some time to gain knowledge in this young field of research. It took some effort, too, to analyze the existing systems and to reason if one of them is suitable to be integrated to our system. Well, after some weeks, the answer was clear... The reasons to reject all programs are shown in chapter A.1. In chapter A.2 we will introduce one program in more detail.

A.1 Existing Programs and Papers

If we want to compare MP3 files by their content and or want to classify a given MP3 file to its genre, we have to analyze MP3 files and extract the most significant features. Since this is a much higher level of analysis, we will deal with this type of feature extraction only briefly in this paper.

With an MP3 decoder we can extract the sound from an MP3 file. The result is a stream of values representing the amplitude at a point in time. This can be regarded as a digital wave, quantized in time and amplitude. So the music is represented as a wave in the time domain. We can transform this wave into the spectrum in the frequency domain by using the Fourier Transform. So we can view the music in the domain in time and in the domain of frequency.

Let us analyze a very simple approach to compare two MP3 files: We could compare the frequency vector of two MP3 files and see, if they are similar. That could simply be a function that computes the distance between these two vectors. If the distance is above a certain threshold, the two MP3 files do not contain a similar song. On the other hand, if the distance is rather low, we could regard the MP3 files are similar or are at least related in that way, that they contain the same type or genre of music.

This first idea sounds logic and easy to implement. But experiments have shown very quickly, that this simple approach is not working. Yet a small difference in the music results in a different spectrum. So, this approach is not working and we would have to design another filter.

In the past decades a lot of research has been arranged in classifying music and comparing digital songs. The results of the experiments are fairly adverse to our project. Music or a song has to be described by many features. The frequency spectrum by its own is a bad indicator to classify songs. At the moment there seems to be no automatic technique in characterizing music in a way, that songs can be compared with satisfactory results.

A couple of experiments and projects dealt or still deal with classifying music. But all of the programs contain some manual work in describing music.

- David Pye [16] researched about the automatic classification of music. The techniques “Gaussian Mixture Modelling” and “Tree-based Vector Quantization” led to good results. A set of MP3 files was used to train the database, and related songs were used to test the

database. Up to 90 % of the music was recognized correctly. Unfortunatly, the data base can only destinguish very coarsely. Six different music genres have been considered during the experiments. Additionaly, the training of the database took a while.

- The Chung Hua University in Taiwan [13] dealt with the recognition of a file to a given peace of music. The user inputs with his voice the melody of a part of the desired song by humming. The system computes the most likeliest song.

  The indexing of the MP3 files of the database is done manually. A human being segments the music into several parts. This is a huge effort can not be done efficiently for a large music database.

- At Pandora [14] the music files are classified completely by humans. An employee of Pandora listens to a song and classifies it by hundreds of criteria. The system computes related songs by the distance of this criteria vector.

  A system user can enter his favorite song and the system will play a song with akin criteria. A more detailed description of Pandora is illustrated in Chapter A.2.

- A similar project is the MusicSurfer of the Music Technology Group of University Pompeu Fabra of Barcelona [5]. A centralized server contains the information of many songs, classified by Timbre, Rhythm, Genre Probability, Danceability, Dynamic Complexity and others. The user can upload an own music file a select the desired classification creteria, the system will then compute a list of related songs. The user can listen to the songs by downstreaming them.

- A framework for analyzing and synthesizing music is jAudio [2] of the Autonomous Classification Engine project of the McGill University of Montreal. One can analyze MP3 files by selected features and store the results in a special XML format. The program is accessible via GUI and via command line interpreter. Unfortunately, the program has some fatal unfixed bugs.

- The MusicMiner [4] is a project of the Data bionics department of the University Marburg, Germany, dealing amongst other with the analysis of music. It accepts a list of MP3 filenames and analyzes the content of the songs. These features are used to calculate the relationships of these to each other. The result is presented in a interesting looking geographical map. So, the similarities of the songs are shown in a very unusual way.

  However, the used similarity routine is working in a coarsely way. So there is some work to do to get satisfying results.

As one can see, there are a lot of promising attempts for feature extraction and classification of MP3 files. But until today, every mentioned system above needs human assistance or cannot applied to a general set of MP3 files. So, there is no fully developed system on the market that can fulfill the desired tasks of this project. So, we kept it simple and created our own functions.

A.2 Pandora: A Music Discoverer

Pandora [14] is an interactive interface in the World Wide Web to discover music. The user inputs the name of a song or a band's name into the system. It computes a song title of the same music genre and this

song is played back through the Web interface. The user can vote, if he likes this song or not, and so the system tries to find out, which songs and genres the user prefers. The system will play songs of the user's favorite music. It consists in a database with more than one hundred thousand songs of thousands of artist. The songs have been analyzed and features of the songs have been collected.

Pandora's basic approach is a completely different one than our Amos II system. The songs of Pandora are analyzed manually. A human music analyst listens to a song an classifies it by the melody, harmony, rhythm, instrumentation, orchestration, arrangement, lyrics, vocals and many more. This process takes an hour for each song. The database contains the title, the actual music and hundreds of genre information for each song. So, the song selection is based on the *manual* classification by music analysts of Pandora.

Our approach is to compare songs *automatically* by their content. The classification depends only by the songs' content with their filtered histograms stored in the database.

So, Amos and Pandora do not have that much in common. However, Pandora is nice program as a personal DJ running in the background.

Figure 6: The GUI of Pandora

Figure 6 shows the GUI of Pandora. An example of a session of the program: After the input "Abba", Pandora classified it as *pop rock qualities, disco influences, mild rhythmic syncopation, a subtle use of vocal harmony and melodic songwriting.* The first song played was "Take A Chance On Me (Live)" by Abba.

Next song was "Fading Like A Flower" by Roxette, because it has *similar mild rhythmic syncopation, subtle use of vocal harmony, a dynamic female vocalist and many other similarities* to the once inputted band name.

Voting *bad* the next song "If Love is Out Of The Question" by Celine Dion, the station switched to "Love In A Vacuum" by 'Til Tuesday with the features *mild rhythmic syncopation, major key tonality, mixed acoustic and electric instrumentation*.

References

[1] Homepage of Audiere. `http://audiere.sourceforge.net/`.

[2] Homepage of jAudio. `http://coltrane.music.mcgill.ca/ACE/features.html`.

[3] Homepage of mpg123. `http://www.mpg123.de/`.

[4] Homepage of MusicMiner. `http://musicminer.sourceforge.net/`.

[5] Homepage of MusicSurfer. `http://musicsurfer.iua.upf.edu/`.

[6] Homepage of the madlib library. `http://www.underbit.com/products/mad/`.

[7] Functional Data Integration in a Distributed Mediator System. `http://user.it.uu.se/~torer/publ/FuncMedPaper.pdf`.

[8] Amos II Release 7 User's Manual. `http://user.it.uu.se/~udbl/amos/doc/amos_users_guide.html`.

[9] K. Salomonsen et al. *Design and Implementation of an MPEG/Audio Layer III Bitstream Processor*. 1997.

[10] Homepage of MP3' Tech. `http://www.mp3-tech.org`.

[11] Homepage of Uppsala DataBase Laboratory. `http://user.it.uu.se/~udbl/amos/`.

[12] David S. Linthicum. *Next Generation Application Integration: From Simple Information to Web Services*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[13] Chih-Chin Liu and Po-Jun Tsai. Content-based retrieval of mp3 music objects. In *CIKM '01: Proceedings of the tenth international conference on Information and knowledge management*, pages 506–511, New York, NY, USA, 2001. ACM Press.

[14] Pandora™. `http://www.pandora.com`.

[15] Vaidrius Petrauskas. *Object-Relational Wrapping of Music Files*.

[16] David Pye. Content-Based Methods for the Management of Digital Music. 24a Trumpington Street, Cambridge, CB2 1QA, UK. AT&T Laboratories Cambridge.