UNIVERSITÄT
DES
SAARLANDES

Spin the Records:
Synthesizing Electronic Dance Music Using Model Checking Methods

Bachelor Thesis in Computer Science

by

Jonas Langhabel

September 2, 2013

supervised by

Prof. Dr. Bernd Finkbeiner

reviewed by

Prof. Dr. Bernd Finkbeiner
Saarland University, Reactive Systems Group

Prof. Dr. Meinard Müller
International Audio Laboratories Erlangen

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides Statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement in Lieu of an Oath

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken, _____ _____

(Datum/Date) (Unterschrift/Signature)

Contents

Abstract

Electronic dance music enjoys great reputation today. However, it is not suitable for simply playing one track after the other, as contemporary media players would do. A new world of sound is entered if tracks are combined in disk-jockey style. This thesis demonstrates how to automatically synthesize an uninterrupted stream of music using disk-jockey techniques. The approach raises a search problem, that is tackled by employing model checking methods. A program capable of synthesizing all legitimate track combinations is designed whereas a temporal logical formula specifies what desirable qualities a combination shall have. The model checking tool SPIN is then used to extract a sequence of tracks that meets the listeners' requirements.

Glossary

amplitude	here: the maximum extent of a sound wave
BPM	short for Beats Per Minute; the standard measurement to describe a track's tempo
CDJ	a CD player used by DJs that emulates the turntable look and feel; brand name of a range of Pioneer CD players
chroma	a quality of a pitch; the chroma is the same for a pitch in all its octaves
DJ	short for disk-jockey, a person who chooses tracks and combines them aesthetically
DJing	what a DJ is doing: euphoniously playing an uninterrupted stream of music; mixing is used as a synonym
e.g.	short for exempli gratia; for example
energy	the energy of a track or part of a track measuring its danceability; high energy motivates people to dance
et al.	short for et alii; latin for and others
fade	to fade; turn the volume up (fade in) or down (fade out) slowly, or to do a crossfade
GUI	short for Graphical User Interface
i.e.	short for id est; latin for that means
key	in music theory keys are groups of notes considered to be harmonious, they are named after their fundamental note, the tone or pitch of a piece of music; a track can be assigned a key
LTL	short for Linear Temporal Logic; a modal temporal logic
MIDI	a communication standard between electronic instruments; short for Musical Instrument Digital Interface
MIR	short for Music Information Retrieval; the science of obtaining information from music

mix	the word originated from the hardware DJ mixer; (verb) to mix/mixing describes the process of blending tracks together, DJing is used as synonym; (noun) a mix is a sequence of tracks that are combined to appear as one continuous track
pitch	the perceived frequency of a sound; pitches are compared to being higher or lower
segue	smooth and seamless transition between two tracks paying special attention to a suitable location for the transition; during a segue usually both tracks are played simultaneously for a period of time
track	the common word for a song when it comes to electronic music; also sometimes called a record
transition	a mix from one track into a new one; also called a segue

Knowledge speaks, but wisdom listens.
Jimi Hendrix

Chapter 1

Introduction

In the 1980s, when first electronic instruments such as drum machines and synthesizers became available, the new genre of electronic dance music was born. This kind of music constituted a novelty, as it was not performed live by the producer but played from records by disk-jockeys (DJs). Soon, DJs started to interact with the music in a more and more artistic way, for example by acting as producers themselves and editing the tracks they play. Thus, they quickly became the main attraction in music and dance venues and still are. Their main field of work takes place in clubs, which today mainly host electronic dance music nights. Besides they work for radio stations or record labels in a studio to create mixed compilations for the 14.6% of Germans above the age of 14 that fancy this music genre [23]. Its worldwide market (based on record sales, events and DJ bookings) for the age group from 20 to 34 is estimated to be €2.7 billion [10]. For a night's work (playing 2 to 5 hours) DJs earn from 200 $ up to as much as 250,000 $, depending on their popularity and experience [28]. Especially for bars, where music is only secondary, they constitute a big cost factor. Today, DJing has run through a digital revolution. Computerized DJ systems already provide a lot of assistance, however, few solutions exist that actually operate without human intervention. While DJing is an artistic job that requires many years of experience and a lot of creativity, the work of DJs still can be automated further. Current media players do a blindfolded fixed duration crossfade between successive tracks. This thesis proposes a novel kind of player that uses DJing techniques to do the transition between successive tracks and aims at replacing current media players whenever electronic dance music is to be played.

1.1 Approach

Electronic dance music tracks generally have long monotonous rhythmic introductions and outroductions intended to be played in juxtaposition by a DJ, to smoothen the transitions between two songs. For this reason electronic dance music is not normally played one track after the other. Common media players (e.g. iTunes) do not play those parts of the tracks in parallel. As a consequence, the intro of the new track succeeds the outro of the old track, and thus, even two tedious parts of the tracks are played in a row. Ideally, we do not want to listen to these 'purely functional' parts of the tracks at all.

This is were the DJ[1] comes in: His ambition is to successively play tracks in such a way that there is never a dull moment. However, there is not always a DJ at hand to eliminate the monotonous portions of the song for the listeners: For example if you listen to your music at home, while being on the move, at private parties or if a bar cannot afford to employ a DJ. Furthermore, the DJ himself relies on some kind of backup if he has to take a break. In all these cases a different way of playing the music is needed if you still want to enjoy the best possible listening experience. In other words, there is a strong need for a music player that automatically combines tracks like a DJ would. This thesis tackles the problem of automating DJing in order to facilitate a better listening experience for electronic dance music when no DJ is present.

The essence of DJing is to appropriately order a given list of tracks and then to play this list while making seamless transitions between the tracks. The process of ordering tracks according to their musical properties and to maximize fit between successive tracks is called **sequencing**. Ordering criteria for tracks are their tempi and keys. Making seamless transitions or segues between tracks (which involves playing two tracks simultaneously for a period of time) is in DJ terms referred to as **mixing**. The resulting uninterrupted stream of music is called a **mix**. A DJ's primary responsibility is to keep his audience dancing. Hence, he chooses the segues in a way that skips musically uninteresting parts or combines them with other parts. In addition to taking care of combining only fitting parts, he aims at making segues last as long as possible, so that alterations happen very slowly. As a consequence, these transitions are not perceived as such and the mix appears seamless.

Electronic dance music tracks feature a *nice* structure. They can be divided into several **sections**, which are the biggest self-contained passages within a track. Each section typically has the *nice* size (i.e. number of bars) of a power of two. Section boundaries can easily be detected by the human listener, as sections clearly differ

[1] Throughout this thesis a DJ is referenced to with 'he'. Even though the DJ business is dominated by male DJs, this is solely done for reasons of simplicity and for achieving a better readability.

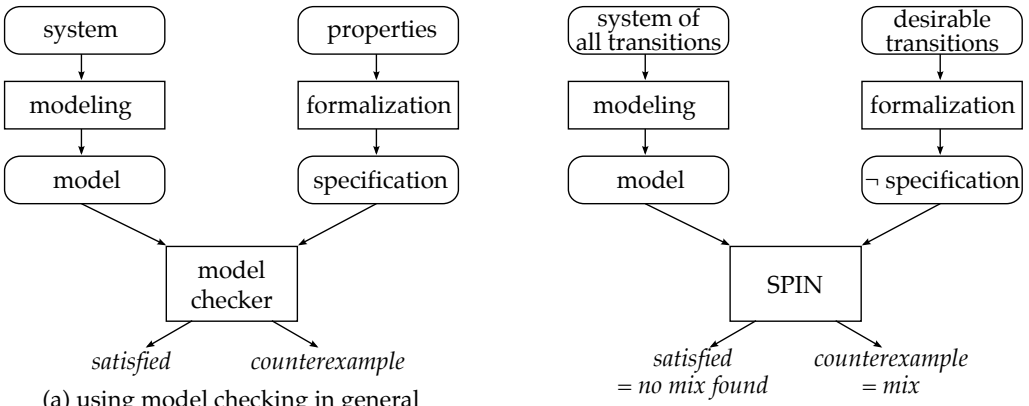
in amplitude and frequency spectrum (i.e. loudness and tonality). Computerized segmentation of a track into its sections is also already a well researched task. As the overall music genre is heavy on rhythmical and percussive patterns and a section mostly consists of — at the most slightly varied — repetitions of such patterns, the structure within a section can be considered homogeneous. A segue between two sections (i.e. two tracks) is then made by playing these sections simultaneously.

For making a transition, the DJ has to find a track in his library that has a section which fits to a section of the track currently playing. He has $n \cdot d$ possibilities to find a pair of sections for the segue, where n is the number of tracks and d depends on the number of sections of the tracks. As one can imagine, there are even more possibilities on how to form a complete mix out of a set of tracks. That is the task of finding a segue n times repeated, which results in $(n \cdot d)^n = n^n \cdot d^n$ possibilities to arrange the mix. We see that the search problem we are going to solve is clearly exponential.

The nodes of the search graph are formed by each section of every track. Basically the nodes are completely interconnected. If we take a transition from one node to the other, it represents a segue if the nodes stem from different tracks. If the sections are from the same track, it just represents normal continuous playback. Yet, not all legitimate paths (i.e. the paths we can take through the graph) describe desirable mixes, meaning that they sound pleasant. The outcome is affected by several factors: Of course, sections of each track should be played one another. Furthermore, the section sizes in a transition should be the same so that all section boundaries are aligned. Also the harmonics and tempi of both tracks should be compatible. Finally, the transitions should respect the overall structure of a track. For example, tracks have sections that build up tension and make the audience anticipate a peak in the successive section; others do the exact opposite and serve as regenerative periods. The DJ might skip a regenerative section, but stealing a musical peak from the audience is not approved of. Therefore, DJing rules are formulated that specify which sections can be combined, depending on the sections' role in the tracks. In sum, finding a mix forms the following **search problem**: We want to find a path through the graph describing all possible combinations, that complies with all proposed desirable properties.

Reactive systems are systems that interact with their environment, by sensing inputs and modifying it if applicable. Computers, robots, washing machines and MIDI keyboards are all examples of reactive systems. If they are complex and in particular if they are critical for safety (e.g. a fire alarm system or a nuclear power plant control system), they need to be verified to work correctly under all circumstances. One method to verify the correctness of these systems is **model checking**. In model checking, the possible system behavior is defined by a model and the properties

this model should show are described by a specification. A model checker is then used to exhaustively check if the given model satisfies the specification (i.e. to do the model checking). If the model is complex, a gigantic number of states of the model have to be considered (often more than 10^8). Sophisticated methods use processing power and memory extremely efficiently and can explore even much larger numbers. As model and specification are given in a mathematically precise manner and the model checker uses mathematically correct algorithms, the results of it guarantee correctness (if model and specification describe the wanted system appropriately). If the model checker encounters a violation of the specification, it returns a so-called **counterexample path**: a sequence of states which leads to the violation. This path reproduces the sequence of states and transitions between those states, which caused the error. Short counterexamples are preferred by humans because they are easier to understand. Then, having the knowledge of how an error is caused, the model can be mended. Basically, model checkers employ powerful search algorithms that try to make the model fail. They search for a violation of the specification; and if they cannot find one, the model is proven to be correct [3, pp. 1-14]. They are applied as illustrated in figure 1.1a.



(a) using model checking in general

(b) how it is used in this thesis

Figure 1.1: Schematic view of the model checking approach

Model checking emerged at the same time as electronic dance music came into existence. As it constitutes an important field of research, it has been studied a lot and features powerful, technologically mature and fast model checkers. In this thesis, we will demonstrate how the search problem stated above can be handled using model checking. A path is a sequence of states and transitions, that needs to give the information what track is playing at which time. Possibly two tracks are playing simultaneously. Therefore, the model has to be capable of playing at least two tracks in parallel. The obvious and natural system in which paths meet this requirement is a DJ setup (consisting of two players and a mixer, as explained

in paragraph 2.2). Each state of our model resembles a snapshot of a DJ setup, transitions to a different state can be made if they can also be made on the physical DJ setup. In other words, the model can execute all moves a DJ can, and thus, the model can create all mixes a human DJ can come up with. Furthermore, it needs to be specified, which mixes are considered sound. The specification is formalized as a temporal logical formula that primarily encodes DJ techniques.

SPIN is the name of the model checker used to generate a mix. It was developed in a group around Gerard J. Holzmann in 1980 and is still developed further to keep pace with the state of the art. SPIN is an acronym for Simple PROMELA Interpreter, while PROMELA is the programming language used to create the model for SPIN [30, p. 3]. The specification is created such that it describes which mixes are satisfactory. Then, the model checker checks whether the model satisfies the specification or not (i.e. whether the model can produce an acceptable mix or not). It will return a 'satisfied' if the specification can be satisfied. If it cannot be satisfied, the model checker returns a counterexample for a mix (one that does not always adhere to the DJ rules). However, simply knowing that a mix exists does not help on the task of finding one. Therefore, we apply the trick of negating the specification (see figure 1.1b). A counterexample now constitutes a path that does not comply with the negated specification. But that means, that it complies with the specification and accordingly with the DJ rules. Thus, the counterexample path poses a valid mix that adheres to the rules from the specification. For a positive run, we get a 'satisfied' that we now interpret as 'no mix found': No counterexample for the negated specification was found, or in other words no example for the specification was found.

Figure 1.2 shows this thesis's approach on how to synthesize a mix: First, the tracks that are mixed have to be divided into their sections. Then, they are saved in a track library. The model is created out of this track library and a DJ system. Besides, theoretical knowledge about DJing techniques and principles are encoded to a temporal logical specification. The model and specification are both given to the model checker, that hopefully will return a counterexample which embodies a mix. In the end, the counterexample can be transformed back to the audio signal that we sought after.

1.2 Contribution

The main contribution of this thesis lies in developing a way to automate DJing using model checking methods. Before, the only link between model checking and DJing might have been that DJ software could be verified by model checking tools.

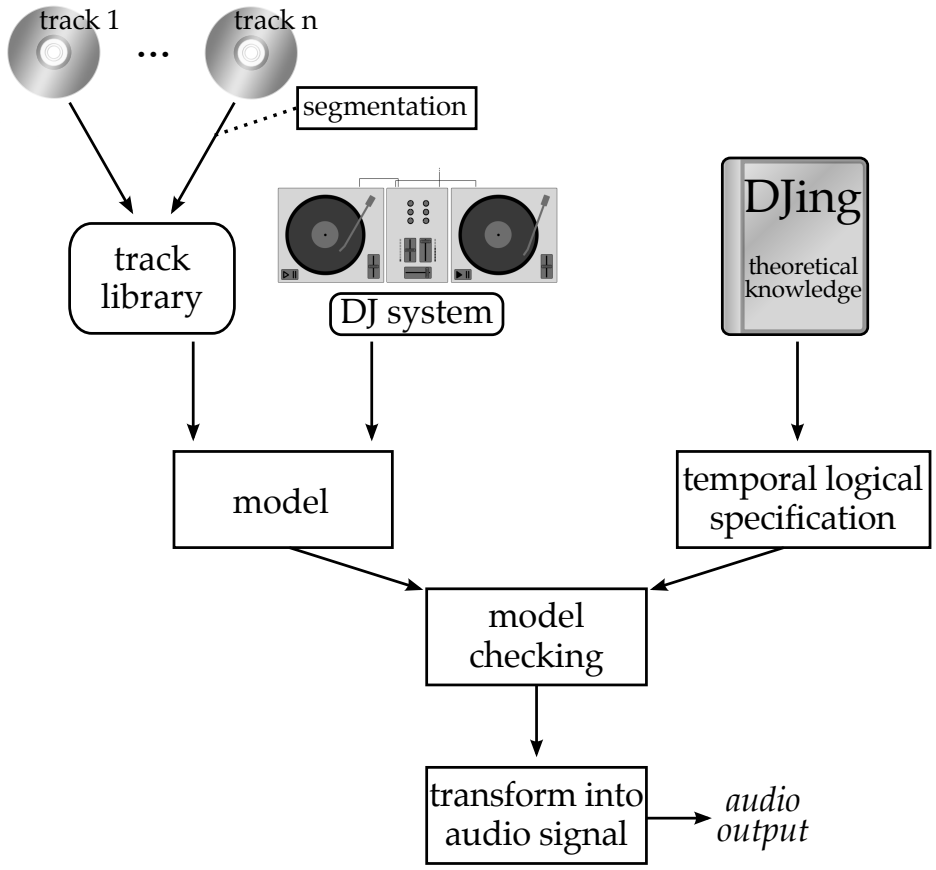


Figure 1.2: The overall synthesis approach

This thesis presents the analysis and automation of the task of DJing, while actually relying on model checking to find a mix.

Model checking methods are applied to a novel and uncommon area in a upside down manner. A DJ setup is modeled in PROMELA. Therefore, the setup's components are inspected and described mathematically. Two alternatives to represent time are implemented: Either time progresses in equally long bar steps or in dynamically changing section sized steps. DJ techniques, rules and principles are analyzed. Then, the techniques and rules are added to the system as constraints and combined to a big temporal logical formula. The model and formula are given to SPIN, which returns a mix that can be played on digital DJ systems and adheres to common DJing techniques.

This thesis pushes the limits of how music can be synthesized in the electronic dance music genre and DJ domain. The borders between what is considered artistic or mechanical in the work of a DJ are redefined by demonstrating that the proposed solution automatically and successfully combines tracks in an analogue fashion

to what a DJ would do. Therewith, the currently limited automation features of DJ software are enhanced. Routines such as choosing a suitable track and finding transitions are automated in a reliable manner.

An application called **autoDJ** is realized, serving as interface for generating customized mixes and playing them by making use of a third party DJ software. It constitutes a fully functional modular constructed basis that can be extended or modified within the scope of future work. Its components are the Graphical User Interface (GUI), the mix creation module that can be adapted to model checking novelties and the MIDI controller that remote controls DJ software very much like a real DJ would. The ultimate intention of this application is to replace other media players for playing electronic dance music.

My first relationship to any kind of musical situation is as a listener.
Pat Metheny

Chapter 2

Background

While DJing is a generic term that is not linked to a specific style of music, most DJs nonetheless have their preferred music genres. In this thesis we will address the issue of DJing for the electronic dance music genre. Before we can discuss how to automate the task of DJing, we need to analyze what it consists of. Therefore, we first need to introduce the basics of electronic dance music. Then, we put the spotlight on fundamental and advanced DJing techniques by also explaining the tools that are required.

2.1 Electronic Dance Music

As we can derive from the name, electronic dance music is made with the help of electronic devices (such as drum machines, synthesizers or computers) and serves the main purpose of making people dance. It is the music genre being played in nightclub settings, that might also be referred to as club music. However, club music — depending on the definition used — covers all music played in nightclubs, which also includes other genres such as hip-hop. Electronic dance music's most important subgenres are house, techno and trance, of which house is the dominant one today. There exist many sub-subgenres of those that differ a lot in character and sound, yet all share the same leading features: Rhythmic percussive elements and a most of the time continuous beat as a driving force. These characteristics also make dancing to this music easily possible. It is typical for electronic dance music in general that it is much heavier on rhythmic than on melodic elements.

To best explain the structure of this kind of music it is beneficial to take a composers point of view. The smallest component in an electronic dance music track[1] and the

[1] When the term track will appear in the further course of this thesis it will always refer to electronic dance music tracks.

basic time measure is a **beat**. Tracks can be vertically divided into several **layers**. Each layer represents for example one instrument (synthesizer, several drums, hand claps, ...) or a singer. The bass drum line is a fundamental layer as it marks the beats in a track. Usually, at least the kick drum sounds on every beat (called four-on-the-flour rhythm). This fact makes it fairly easy to detect the tempo, that is then noted in Beats Per Minute (BPM). The tempo of house music usually varies between 118 and 135 BPM, yet today's tracks nearly all have 128 BPM. During the course of the track layers are added, removed or altered. A **bar** is the smallest coherent unit out of which a track is constructed. Each bar has four beats. The first beat in a bar is called the **downbeat** [18, pp. 160-164].

Electronic dance music is very ordered. The overall structure consists of several **sections**. Each of these sections should be thought of as a uniform contiguous entity that represents a complete musical idea and sounds quite homogeneously. However, in the course of the track each section serves a purpose, and thus, can not be regarded independently from neighboring sections. For creating a section the composer will start off with two bars. These two bars are duplicated, but the end of the new fourth bar is varied slightly. Again, these four bars are duplicated and the eighth bar is varied in a more noticeable way. A 16 bar sized section is obtained by once again duplicating this eight bar phrase, yet without varying the last bar. The resulting section has slight variations every four bars and a noticeable one every eight bars. Bigger sections are a multiple of two of the eight bar blocks. Furthermore, every eight bars new instruments might be added, such as hi-hats, hand claps, snare drums or synthesizer melodies. The track is then composed out of such sections and their duplicates [26]. This repetitive structure makes tracks predictable and as a result you quickly become familiar with previously unknown tracks; a characteristic that explains why this music enjoys such a popularity in dance venues (as people prefer to dance on familiar sounding music). There are several section types that we can distinguish:

- The first section of a track is called the **intro**. It consists of a simple drum pattern that gets extended slowly by more sophisticated layers during its course. Depending whether this track was composed for being used by DJs or for being played in the radio, this section tends to be longer or shorter (the resulting different versions are called DJ and radio edits).
- After the intro the main part of the track starts. Half of the main part consists of **bridge** sections in which tension is either created (**build-up**) or resolved in a regenerative part (**interlude**).
- The other half of the main part consists of the high energy sections, to that the audience wants to dance. These are called **chorus** and are characterized by

their high average amplitude.

- The last section of a track is called the **outro**. It is arranged similar to the intro, but the other way round. That means during its course layers are removed until only a simple percussive pattern is left. Its length depends on whether the track is a DJ or radio edit.

Intro and outro carry only little harmonic information so that a DJ can play the outro of one track simultaneously to the intro of another, without causing disharmonies. It is convenient that sections typically have a number of bars that is a power of two. As we will explain in detail later, during a segue two sections are played in parallel and those sections are required to be of equal length. The number of bars in a section — in the following also referred to as **section size** — has to be one bar at least whereas the most common section sizes are 8, 16 and 32 bars. Finding a segue requires to find a pair of equally long sections. That we actually can find such pairs just becomes possible with the little choices the power of two offers the section size to be.

2.2 Analog DJing

The lack of melodic content in intros and outros causes them to appear quite monotonous and uninteresting, which is likely to stop people from dancing. A DJ has to spice things up to prevent people from wanting to leave the dancefloor. Therefore, he plays intro and outro or other sections of the tracks simultaneously and makes the results appear as one continuous new track, the mix.

To be able to play two tracks at the same time a DJ needs special equipment:

- In analog DJing vinyl records are used, that are being played by at least two **turntables**. DJ turntables have controls such as a play/pause button to start and stop playback and a linear potentiometer to adjust playback speed (called the pitch control slider). It allows to smoothly change the tempo within a range of about ±10%. Increasing the tempo by 6% also raises the key one semitone up [18, pp. 245-246]. To play a particular part of the track, the turntable's tonearm can be moved to the respective position on the record.

- The turntables' analog audio outputs are connected to a multi-channel **DJ mixer**, which is a slightly modified mixing console. Each channel (here called the left and the right channel) has a separate volume control and an **equalizer**. Equalizers consist of a set of small circular potentiometers used to amplify or weaken limited frequency bands (such as 'low', 'mid' and 'high'). The altered

channel signals are then combined to the main output channel. Moreover, the volume controls are supplemented by a **crossfader**; a linear potentiometer that could basically replace both volume controls and can be controlled using one hand only. With it, one channel's volume can be faded out (i.e. decreased) while simultaneously the other channel's volume is faded in (i.e. the volume is increased). In the leftmost position the output signal solely consists of the left channel's signal, in the rightmost position solely of the right channel's signal. In between it consists of a mixture of both input channels with a ratio that is defined by the crossfader-curve (see figure 2.2). Another feature a DJ mixer has, is the possibility to listen to one or both channels on headphones independently from the main output.

Two turntables and the mixer form a DJ setup. What each component looks like and how they are typically arranged is illustrated in figure 2.1.

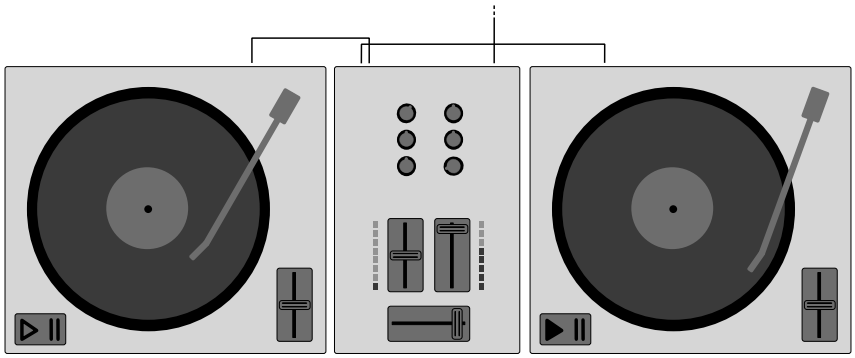
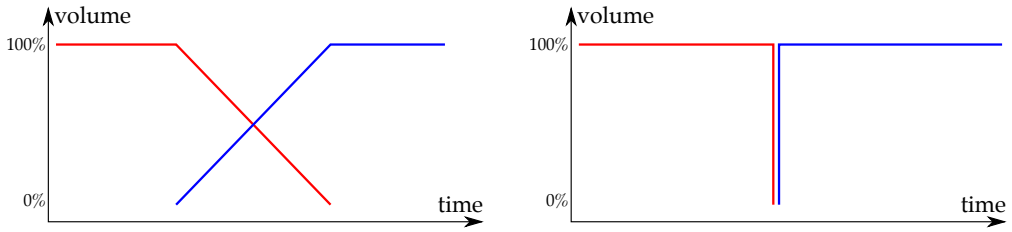


Figure 2.1: A typical DJ setup

A DJ is assessed according to his ability to mix two tracks seamlessly together, so that the transition is heard as little as possible. Basic techniques to do that are the blend and the cut. A **blend** is a period lasting as long as possible, in which both tracks are played in parallel. During that period, the crossfader is used to slowly fade from one track to the other and also the sound is adjusted with the equalizers. The composition of the output channel out of both input channels is illustrated in figure 2.2a. This graph shows the volume changes depending on time and constitutes the so called crossfader-curve. Each of the two channels (decks) is represented by a different color. The volumes in this curve might also change according to different arbitrary functions. The slow fade in and out is the gold standard for electronic dance music. In hip-hop for example, the **cut** is a popular technique. In it the crossfader is slid from one end to the other in one quick movement. The aim is to replace the track instantaneously by another one, which is useful for tracks that would disharmonize if they were played together. The output channel's composition during the cut (its crossfade-curve) can be seen in figure 2.2b.



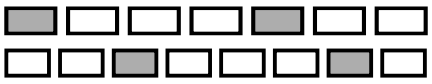
(a) The basic X-fade (b) The cut

Figure 2.2: Crossfading curves

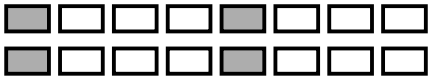
Both techniques require preparations: First, the tracks need to be **tempo matched**, which requires alteration of the tempo of one or both turntables. The easiest method is to adjust the tempo of one turntable until both tracks are played equally fast. Figure 2.3a illustrates two tracks played at different tempos; each box represents a beat of a track. Secondly, the tracks have to be **beatmatched**. Two tracks played at the same speed but being not beatmatched produce no good sounding results; they *clash*. In the process of beatmatching tracks are aligned so that the basic drum patterns sound simultaneously. This process is also known as synchronizing. To achieve that, the DJ slows down one track by manually braking the record or speeds it up by giving it a push forward. However, this alone does not suffice to completely align the patterns. As described in chapter 1, a bar is the smallest coherent unit in a track. To not mash up the percussive elements, the DJ needs to further align the downbeats of each bar as they mark the start points of these elements (see figure 2.3b, the dark boxes mark the downbeats). The displacement of the downbeats is called the **phase difference**. Furthermore, the DJ should align the sections of the tracks, such that they start at the same time and have the same sizes. Both is done by pausing the new track at the starting point of a section (which implies that it is also the starting point of a bar) and by waiting for the exact moment when the correct section in the other track begins, to start it. This technique is called **section matching**. Then, the complete patterns are aligned and the groundwork for a seamless mix is accomplished [18, pp. 160-164, 241-245].

In the process of **sequencing**, the DJ has to plan which track follows the other. A list of tracks that has intentionally been put in order by the DJ is called a **set**. This task includes an artistic component of arranging tracks in a thrilling order and a mechanic component of avoiding to have successive tracks in incompatible keys and of ordering tracks in a way in that the tempo varies smoothly (and systematically) during the set.

All the processes described require a lot of practice and even more proficiency if they have to be performed quickly. However, they have a strong mechanical character.



(a) Tempo not matched, beats and downbeats not aligned



(b) Beatmatching done correctly

Figure 2.3: Schematic view of beatmatching

Computerized systems have been developed to assist the DJ in these tasks or that even fully automatically take over a sub-task.

2.3 Digital DJing

While analog DJ equipment nearly stayed unchanged until today, digital DJing has been subject to ample innovation since it first gained importance in the 2000s. Similar to other products of the electronics and computer sector new products and features are being developed continuously. Remarkable digital DJing products today include Pioneer's *CDJ* hardware players (since 1994), and the DJ software solutions *Traktor* by Native Instrument (since 2000), *Virtual DJ* (since 2003) by Atomix Productions, *Scratch Live* by Serato (since 2004) and their open source equivalent *Mixxx* (started in 2001 for a doctoral thesis) [36, 29, 33, 39]. These DJ software solutions are developed in close collaboration with DJs as they chiefly pose a human-computer-interaction problem. For example Mixxx also serves as testing basis for new haptic DJ interfaces and aims at being easily and intuitively operated by DJs who have only had experience with analog DJ systems [2]. For the sake of technical progress, portability and an extended functionality, developers strive to replace analog DJ equipment with software solutions and/or digital hardware devices. It started off with Pioneer's CDJs, standalone DJ CD players replacing turntables. Today, Pioneer's CDJ players very much resemble computers with smaller displays and a user interface that is designed to simulate turntables. Furthermore, computers are used to run software that can accomplish everything an analog DJ setup is capable of, or even more. These software solutions can either be controlled with standard input devices such as keyboard and mouse, or with so called *MIDI controller* hardware devices that aim at creating the same operating experience that is achieved by using turntables and a mixer.

What is common in all solutions is that the sound is processed by a sound card or

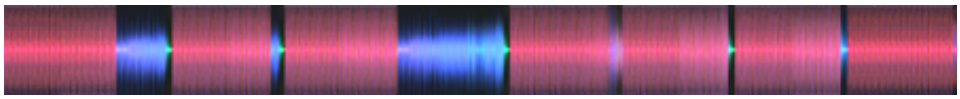
audio interface and that the tracks are stored in a digital format. In the following, we will limit ourselves to considering software solutions only. What previously was called a turntable or player is now called a **deck**. Decks are virtual objects that offer all modification possibilities a DJ turntable has. With the help of a file browser a track is loaded into a deck, then the start position in the track can be moved to wherever wanted. Its tempo can be changed and the digital mixer also features an equalizer, volume sliders and a crossfader. Stretching algorithms (such as [35]) are used to change the tempo at which a track is played. By using a fast fourier transform and manipulating the obtained data, every modification of the tracks that has been commanded in the mixer section can be achieved [42]. While the GUI can show the same information as the analog user interfaces, it can not be controlled effectively using keyboard and mouse only. Digital hardware devices, which try to simulate the sensation of handling analog turntables and mixers are used to overcome the limits given by keyboard and mouse (e.g. CDJ players, combined digital and analog mixers and timecode vinyl systems[2]). These hardware controllers are connected to a laptop and then usually communicate with the DJ software via MIDI. Every button click or fader move is encoded as a MIDI message that can be understood by the software.

On the one hand computers make the task of DJing much easier (as they provide a range of aids) and thereby devalue the painstakingly acquired skills of traditional DJs, but on the other hand they clear the way for more complex maneuvers and creative combinations that even the most skilled traditional DJs were not capable of. A good DJ today does not simply string tracks together: Instead of just making segues his goal is to form something completely new. Hence, he tries to play each track as short as possible, while maximizing the time where several tracks are played in parallel. Thus, DJing today can be seen as doing live remixes of tracks while the borderline between performing and producing becomes blurred.

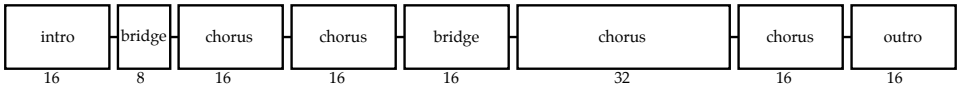
One simple aid of digital systems is that they can display the **waveform** of a track. That is a temporal plot of the track's amplitude. The x-axis represents the time, while the y-axis represents the amplitude and the color coding depends on the frequency spectrum at each point in time. An example waveform representation of a track can be seen in figure 2.4a. The segmentation of this track in its sections is shown in 2.4b. Along with the waveform we will introduce **energy curves** of tracks as seen in figure 2.4c: This curve basically follows the maximum amplitudes at each point in time. Later in this thesis, energy is used to describe the intention of a section. If one section has high energy, then its intent is to make the audience

[2]Vinyl disks that contain no music but digital timecode data are played with standard turntables. The output is routed to special hardware that uses it to calculate the stylus's position on the disk and sends that information to the DJ software.

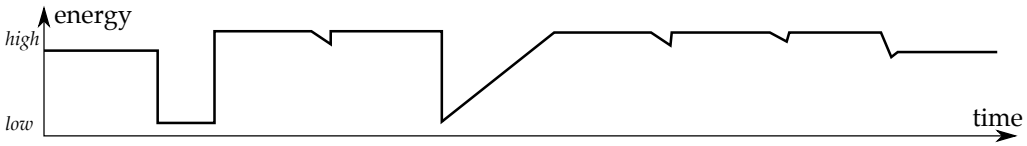
dance; if it has low energy it has a low danceability and serves as a regenerative period. Also energy can be built up in the build-up sections. Now, as the amplitude facilitates the detection of a track's sections (as described in paragraph 4.1), the DJ instantaneously learns the structure of a track and can easily navigate to every position and section beginning. Also the dominant frequencies differ from section to section and therefore, a color change indicates a section change. That further helps the DJ to match fitting sections for a transition. Another aid provided by DJ software today is **automatic synchronization** between the decks. If this function is activated, the software automatically tempo matches and beatmatches tracks. This alignment is on beat level only, meaning the DJ is still required to align phases (downbeats) and sections. When the playback speed is changed on analog turntables this also results in a change in the observed key. The algorithms used to play digital tracks at a different tempo can simulate this change in key or — as this change is mostly unwanted in electronic dance music — can keep the key constant while the tempo is adjusted; this is called the **key lock** feature.



(a) Waveform display of the track Riverside by Sydney Samson (screenshot of Traktor Pro [33])



(b) The track's segmentation; section sizes are given in bars



(c) A sketch of the energy curve of this track

Figure 2.4: Representations of an example track

2.4 Harmonic Mixing

The discipline of mixing only harmonically fitting tracks together is called **harmonic mixing**. Paying attention to the key relationship between tracks enables the DJ to not only combine the tonally poor parts of tracks but to make segues theoretically out of any parts. In a computerized analysis each track is assigned a musical **key** (tools that perform the analysis are e.g. [41] and [33]). A key basically is a set of notes that play well together. There are 24 keys, 12 major and their 12 related minor keys.

Notes of a key's set of notes are used to compose melodies and chords; notes that are not included disharmonize with those that are included. During the analysis the notes occurring in the track are extracted and then the track's key can be derived by finding the best match with a set of notes belonging to some key. Problems arise if not all notes of a key are played. This is not a rarity as many electronic dance music tracks contain little melodies. Then, the key can not be defined unambiguously. Also the key might change in the course of a track, either because the composer wanted to achieve an artistic effect or because he did not pay too much attention to music theory while creating the track [24].

The **circle of fifths** provides an easy method to decide if two different keys harmonize with each other (i.e. if tracks with those keys can be played in juxtaposition). The sequence of keys is ordered so that each neighboring keys differ by seven semitones. Seven semitones correlate to the musical interval fifth, which is said to be the second most consonant musical interval except for the octave. As the octave is perfectly consonant, the sequence of keys can be united to a ring by overlying the octaves, the circle of fifths. Major and minor keys are arranged in different circles, yet in a way such that each major key is adjacent to its parallel minor key. The **camelot wheel** poses a slightly easier to use equivalent of the circle of fifths [20]. Each musical key is assigned a number between 1 and 12. To minor keys an A is added, to major keys a B. Thus, for example C Minor is assigned 5A in the camelot scale, as shown in figure 2.5. To each key given in camelot scale one can find different keys that are in a harmonic relationship by adding 1 to the key or subtracting 1 from it (e.g. 5A is compatible with 4A and 6A). Also the parallel major or minor key is compatible, which is found by swapping A with B or vice versa (e.g. 5A is compatible with 5B).

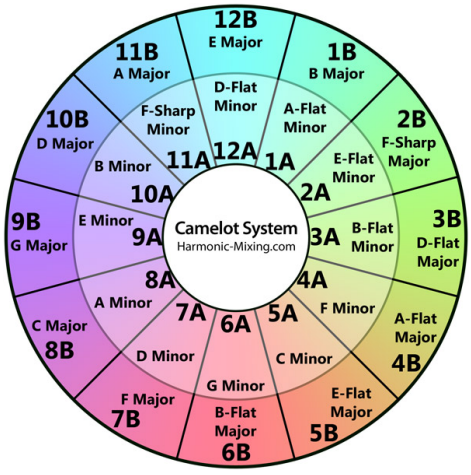


Figure 2.5: The camelot extension to the circle of fifths (source: [41])

Energy boost mixing is a different way of purposefully sequencing a mix according to the tracks' keys. Here subsequent tracks are ordered to differ by a one or two semitone interval (+7 or +2 in the camelot scale). This method aims at increasing the dynamic of a DJ mix as completely harmonic mixes might lack in creating suspense. However, tracks in energy boost mixing are not in a harmonic relationship, and thus, it should be avoided to play two tracks in parallel for longer periods in parts with rich melodic content. Combining the tonal poor sections intro and outro can be considered to be safer [18, pp. 253-255].

As harmonic mixing scales the number of possible successor tracks down it is a powerful tool to help DJs work with huge track libraries. But how reliably do key detection algorithms work? In [24], the accuracy of key detection tools has been tested by comparing the algorithmically calculated key to the manually determined one. The best contenders achieved 42% correct detected keys only. However, if the erroneous results stem from that the key cannot always be assigned unambiguously (as mentioned above) this does no harm. If several keys are possible and one is arbitrarily chosen, then as a consequence the number of possible fitting tracks is reduced. Yet all possible fitting tracks stay in a harmonic relationship, as a subset of a key's note set will still be consonant with another key, if the whole set is. Nonetheless many combinations of tracks will be falsely believed to be harmonic. Thus, we should avoid blindly combining two melodies or sections of rich melodic content, even if both tracks are in a harmonic relationship.

Without deviation from the norm,
progress is not possible.

Frank Zappa

Chapter 3

State of the Art

Today ideas taken from DJing have entered the market of media player software. All major players such as iTunes, Windows Media Player or Winamp feature a crossfade functionality to allow continuous playback. They simply do a blindfolded end crossfade for a fixed duration. These players however are not able to beat and tempo match tracks and are therefore unable to perform proper electronic dance music segues. DJs also have to do the sequencing for the tracks. The concept to always play an — according to various criteria — fitting next track has been adopted by so called music recommendation systems. Noteworthy solutions are Apple's Genius and Echonest (see [40]), which use musical genres and sonic characteristics as criteria. These tools help to keep track of the huge music libraries that are standard today by extracting suitable playlists, very much like DJs preselect tracks to include in their sets.

All prime DJ applications provide a so called **auto mode** that autonomously does crossfades while playing playlists. The synchronize function of today's DJ software enables proper transitions to be achieved, yet the crossfading time still has to be fixed in prior. And as it is preset once for all segues, it is independent of the tracks' structures. One more useful feature they provide is based on L'Hopital's idea to set markers (a flag that bookmarks a point in time) in the tracks, as explained in paragraph 3.1. These tell the software at which point in each track the segue is started.

This chapter lists the leading projects that ambitiously try to replace human DJs. First, academic approaches to the task of automating DJing are introduced, then some existing commercial solutions that claim to play music like a DJ are presented.

3.1 Academic Approaches

Automatically synthesizing music is a popular challenge in computer science, artificial intelligence and musical theory. But to be able to do the synthesis, machines need a concept of what music is and a way to represent it. Humans are quite proficient in dealing with 'out-of-time' musical notations such as sheet music. The term 'out-of-time' means for representations, that they have been written down in a specific notation of time, but now can be interpreted and played using any other timing. In contrast, 'in-time' representations are used by systems that interact with or actively listen to music. These representations for example describe the history of a sequence of notes that has been perceived with their exact timings. Thus, from these the system can derive its current actions and possibly an anticipation of the future. Alan Marsden proposed to specify such an 'in-time' representation with modal temporal logic in 2007 [17]. He analyzed the advantages and drawbacks of several temporal logical operators in discrete and dense time to describe the history of what the system has heard. Moreover, he demonstrates how these formulas can be applied. Thereby, he adopts the low abstraction level of considering single notes. For example a sequence of the notes a, b and c could be described as follows: $S(S(A,B),C)$, where logical propositions X will be considered true at all times the note x sounds and S is the dense time *Since* operator defined as $S(A,B) \equiv \exists t((t < now) \wedge A(t) \wedge \forall s((t < s < now) \rightarrow B(s)))$. Intuitively $S(A,B)$ means that B has been true since A. This formula specifies the order of occurrences of a note, however, not the time interval between those occurrences. Therefore, an oscillator is added to the definition of the temporal logical operator S, that oscillates aligned with the beats in the track. Now, the logical propositions can be claimed to be solely true within certain ranges of the oscillation curve. Thus, the time interval between the occurrences can be specified, while conveniently a small range of imprecision is accepted (at least little imprecision in timing can not be avoided when music is played). Yet, Marsden's work is on a too low abstraction level of music for the purposes of DJs; they rather look at a section level of abstraction.

In 1998, even before the breakthrough of digital DJ software, Christophe L'Hopital proposed his idea of how to automatically perform segues between two tracks [31]. His approach requires to by hand predetermine two markers in each track: The first indicates the position in the track where it is started, the second the point where the start of the next track is triggered. When the track is played, these markers signal to the player when to fade in the track and when to fade it out. A segue always begins with the first track playing. When the first track reaches the end marker, the second track is started at its start marker. Thereby the fade in and fade out is done simultaneously, which basically constitutes a crossfade. After the transition

region, the first track is stopped and only the second track continues to play; the segue has been completed. L'Hopital's concept is still used in DJ software in the auto mode. It can be considered to be a working technique that produces well sounding transitions — probably that is the reason why it is still being used, despite all information technological progress — however, it does not specify how long a transition should be and it does not take into account the problem of choosing a suitable next track. In fact his method does not even automate the task of DJing. It rather constitutes a way of preprogramming a DJ mix that can later be replayed by a special player.

Two years later, Dave Cliff of HP Laboratories Bristol announced the idea for his automatic DJ system. By 2005 he accomplished a fully automatic system called **hpDJ**, that has even been tested in a London nightclub setting and successfully tricked 37% of the invited audience to think actually a human DJ was playing to them [6, 7]. His approach goes so far that it can monitor the dancers reaction to the music, and thus, it is able of choosing an appropriate next track, based on its experience what people like. So, very much like a real DJ it senses whether the audience enjoys the music being played and if not it plays different tracks. Furthermore, it is able to compose entirely new tracks. A genetic algorithm is used to modify the composition of the track, while the audience's feedback serves as a fitness measure. Cliff's approach is optimized for electronic dance music. Given a set of tracks, the program lets the user choose whether the set should be sorted by hand or fully automatic. The criterion for the automatic sequencing is the tempo, but as mentioned above the sequencing might be dynamically changed depending on the audience's feedback. A tempo curve can be specified and the tracks are sequenced according to that (e.g. the curve might look like: Start the mix with a low tempo of 100 BPM and then linearly increase it during the mix to 140 BPM). After the sequencing is completed, during the playback segues are done automatically between successive tracks. The overall prespecified playtime of the mix is equally distributed amongst the tracks to be played. Therefore, basically L'Hopital's idea is used to add markers to the tracks: The markers' positions are determined automatically so that they set the part of the tracks that is to be played. As those markers are not set depending on the track's structure or musical conditions, in a reparation step heuristics are used to move them to close section borders. Also the length of the crossfade does not depend on musical conditions. Its length is either a prespecified period of time, a fixed percentage of each track's length, or it is a fixed percentage of the prespecified duration of the mix. During the crossfade, hpDJ applies sophisticated equalizing and filter methods. Both tracks are analyzed for which frequencies do not harmonize with each other and then a customized new equalizer is created that is capable of weakening exactly those frequencies. As analog and digital DJ systems typically only

provide a fixed number of equalizers having an unadjustable frequency spectrum, Cliff's equalizing techniques could achieve even better results than human DJs. Finally, the resulting mix is either saved as an audio file or played live. Faults might occur in the tempo matching process, as Cliff reported accuracy problems due to imprecise tempo detection algorithms. This approach focuses on tempo as criterion to control the sequencing. But tempo is just one factor to decide if tracks fit. Thus, it might happen often that no good segues exist between two successive tracks (this is generally a problem if the sequencing is done in a separate step). Still hpDJ would force that segue to be done as this approach does not consider musical conditions. In a nutshell, the sequencing does depend on the insufficient criterion tempo, and start points of segues as well as the length of segues are determined without considering the tracks' structures.

Hiromi Ishizaki, Keiichiro Hoashi and Yasuhiro Takishima considered the task of automating DJing in 2009 [14]. They focused on finding a solution that as they say 'preserves user comfort', by addressing the issue of how discomforting listeners perceive changes in tempo and degrading tempo quality caused by these changes. Adjusting tempo becomes necessary if you want to do a segue between two tracks having different tempos. Therefore, they determined the boundaries within those the tempo is safe to be varied experimentally. In a preprocessing step, their automatic DJ application creates a database for tempo and beat extraction data of a set of tracks. The tracks to be mixed are chosen automatically from the database depending on their similarity to a query track. Thereby Music Information Retrieval (MIR) is used to test for content-based similarity between pairs of tracks. For calculating the sequencing, the software uses the authors' findings to choose tracks which have a pleasant difference in tempo. To minimize the difference between the original tempo and the playing tempo, they do a dual tempo adjustment: That means they first smoothly adjust the tempo of the first track to a value between both tracks' tempos, for that the discomfort level of the speedup of one track and the slowdown of the other track is equal. Then, while keeping the tempo of the first track constant they perform a tempo matched crossfade to the second track. After the crossfade, the first track is stopped and the tempo of the second track is smoothly altered to its original tempo. The periods in which the tempos are adjusted are fixed to five seconds, and thus, do not depend on the tracks' structures. However, their crossfading algorithm features a variant of downbeat matching. Optimal transition areas in tracks are not discussed. Therefore, their approach can be considered to be an optimized version of a tempo adjusting player that is capable of generating its own playlist. Especially as tempos do not vary a lot in electronic dance music, we can consider their tempo adjustment algorithm to be merely an optimization of a minor subtask of DJing. Furthermore, vital tasks such as how to locate optimal parts in the tracks for the

segues are not looked at.

Composing soundtracks for movies or video games is mostly done by hand and therefore cost intensive. Several approaches aim at automating the creation of soundtracks by combining music samples or tracks on an algorithmic basis. When the setting or mood in the visual material changes, a smooth transition has to be done to an adequate passage of music. This task is closely related to that of automatic DJing as in both challenges a continuous stream of music is synthesized by blending tracks or samples in a euphonious way.

In 2009, Heng-Yi Lin et al. ([16]) tackled this problem of stringing together music tracks from a database by using the concept of **similarity matrices** to detect similar-sounding parts in tracks (as it was first used in [8]). Therefore, features such as loudness, chroma, rhythm and tempo are extracted from the tracks and serve as similarity criteria in the following. These values for each part of one track and each part of another track are compared and the distance is plotted to a matrix. Then, the diagonal window (it is diagonal to match subsequent beats in both tracks, i.e. the time between the compared parts passes equally fast) with the maximum average similarity is the optimal transition area. Finding an order for a set of tracks that maximizes the overall similarity constitutes a search problem. In a preprocessing step, tracks that are dissimilar to more than half of the other tracks are removed from the database. A greedy algorithm is used to find an optimal path, however, it does not necessarily find a global optimum of maximum similarity for all segues.

Stephan Wenger and Marcus Magnor use a special case of the similarity matrix: A self-similarity matrix that compares each part of a track to all its other parts [21]. They look for positions where they can seamlessly jump to another position in the same track. For soundtrack generation this is useful for shortening or extending tracks. Furthermore, it can be used for finding parts that can be looped or even the track can be played forever as done in [32].

Tristan Jehan also proposes to use similarity matrices for finding transition areas that resemble each other in rhythmic patterns [15]. Even downbeats can be aligned with similarity matrices: Specific heuristics can be used to rate unaligned beats to have a lower and correct alignments to have a higher similarity. However, his method does not hinder two tracks with incompatible keys from being mixed and it does not consider structural conditions of the tracks.

From a musical point of view, automating DJing using similarity matrices to find ideal transition areas has the following drawbacks: Two tonally rich tracks might be similar but if played in juxtaposition the result will sound overloaded. The same is valid for passages that include vocals or instrumental melodies. However, if these two cases are excluded we still encounter the opposite problem: Two tedious simple

Name	Price	Since	Sync	Auto Mixing Technique
MegaSeg Pro	199$	1999	✓	crossfading
OtsAV Radio	899$	1999	✓	enhanced end crossfading
BarJock	weekly	2010	✓	crossfading, announcements/ promotional voice-overs
Mixtrax	0$	2011	✓	crossfading, DJ effects, chorus detection, intelligent transition points
DJ Software	100$ - 329$	2000	✓	L'Hopital

Table 3.1: Comparison of Commercially Available Products

parts might be combined, but the result would still be unexciting, although the DJ should eliminate those parts. It would be preferable to play an intro together with a chorus or bridge, which probably differ a lot. The idea behind DJing electronic dance music is not to just match the most similar parts but rather to alter rhythms, to replace layers, to add variation and thus, to create tension. A segue that is done by adding a new layer to the current track will not result in high similarity values. The new layer will most likely be different to what is currently being played. Thus, such segues cannot be found by this approach. But still adding a new layer constitutes the standard way to make segues. As the audience often does not know if this new layer belongs to the current track or a new one, such segues cannot easily be detected even though the parts combined are not similar. However, similarity matrices might return good results for many other genres of music than electronic dance music.

3.2 Commercial Solutions

Several commercial music players exist that advertise an 'automated DJing' feature. This promise, however, should be treated with skepticism. What usually is hidden behind it is that the programs can automatically play a playlist while doing some kind of beatmatched crossfades between the outro and intro sections of successive tracks (so called end crossfading). These segues are not determined depending on the tracks' structures and furthermore the duration of the segues is fixed. Some applications also try to improve their end crossfading techniques, but the automatic DJ mode is clearly not the main focus of their applications but just a supplement.

Table 3.1 gives an overview of some greater scale solutions. The column name 'Sync' is short for synchronization. All listed tools support tempo and beat matching. *MegaSeg Pro* and *OtsAV Radio Broadcaster* are DJ tools intended for usage in radio stations. They feature a fully functioning mixer and decks like a DJ software. Moreover, they feature a scheduler for prearranging radio shows. In addition to that, MegaSeg Pro is also designed to control lighting effects, which further makes it predestined for usage in clubs and bars. While MegaSeg Pro uses basic crossfading

in its automatic mode, OtsAV advertises an 'unexcelled transitioning engine', however, it does not go further than end crossfading: The tracks' start and end parts are analyzed, depending on the gained information an appropriate crossfader-curve is chosen, and then, the tracks are crossfaded for a suitable dynamically calculated duration [34, 27].

BarJock is a music player and a promotion broadcasting tool. Here the DJ can be found in a remote studio doing voice-over advertisements (such as promotional drink offers) ordered by the bar owner, that then can be played. DJing is reduced to the in club settings uncommon method of cheering people on, using a microphone. The technical part of the DJ's job is not looked at. BarJock and its services are paid for in weekly rates that depend on its usage [38].

DJ Software is a place holder for *Virtual DJ*, *Traktor* and *Scratch Live*. These are pure DJ software solutions which are intended to simulate DJ hardware and to constitute powerful tools for the use by professional DJs. Therefore, their automatic mixing method constitutes merely an extra: All solutions allow automatic beatmatched and crossfaded playing of a playlist. What is more, at least Traktor fully implements L'Hopital's idea.

Pioneer's *Mixtrax* is the outstanding solution in this list [37]. It is a grand scale automatic DJing tool that actually tries to be artistic. The software can be downloaded for free, as Mixtrax mainly is a feature of a whole new generation of Pioneer products such as car receivers and portable audio systems. Personal preferences for the kind of transition and the time a track is played can be entered. Mixtrax first does the sequencing according to musical attributes such as tempo and danceability. Then, it tries to detect the chorus and searches an adjoining cut in the structure. This point serves as the start point of the segue. So basically the start points of the segues are aligned with section boundaries. After that, one DJ effect out of a range of effects (for example a flanger, delay, reverb) is chosen to decorate the crossfade. The default preferences favor short segues which are dominated by the DJ effect; the tool can be customized to do longer segues, however, it can only do segues of a predetermined duration. In a test, version 1.3.1 showed a flaw in the beat matching algorithm as tempo was matched, but the beats were not aligned properly. The transition points that it finds are good, however, it in fact prefers to make really fast crossfades. In contrary human DJs aim at playing both tracks simultaneously for as long as possible so that the segue cannot easily be noticed. Mixtrax tries to hide the disharmonies that occur in such short segues by making excessive use of effects. Of course, these kind of transitions can easily be identified by the listener. Excessively using effects leaves a slightly bitter taste, as classical DJing went without effects at all and still clean and proper transitions are demanded to be done without effects. Yet, Mixtrax is the only solution that risks to automatically make transitions between other sections than intro and outro.

> The real danger is not that computers will begin to think like men, but that men will begin to think like computers.
>
> *Sydney J. Harris*

Chapter 4

Musical Considerations

This thesis proposes a DJing technique that will be called **natural mixing**. It is called natural for two reasons: It respects the musical structure and imitates how human DJs work. Compared to the approaches discussed in chapter 3, natural mixing constitutes a novel way to tackle the automation problem. In this chapter, we demonstrate that natural mixing poses an intuitive, powerful, yet computationally complex approach. Chapter 5 shows how it nonetheless can be implemented using state-of-the-art technology. Possibly, distinct human DJing techniques can be adopted as role models in the development of natural mixing. Here, it is geared to the classical analog DJing technique augmented with harmonic mixing. The main focus lies in correct section matching.

4.1 The Segmentation Algorithm

Sections constitute the largest entities an electronic dance track might be divided into. They are musically complete — yet not independent of each other — building blocks of the track. And they have the nice property that they have a length being a power of two bars. Exceptions apart, this holds for almost all sections in every track. The origin of this mathematically nice and clean composition is the way this music is produced; by duplicating building blocks as described in paragraph 2.1. Sections can be distinguished by their energy (i.e. average amplitude or danceability) and intention in the track's overall picture. A chorus's intention is to make people dance, therefore, it has a distinctly higher average amplitude than other section types. In contrast bridges have a lower average amplitude, which allows people to recover briefly. Put in a nutshell, choruses are the high-energy sections and bridges the low-energy ones. During a build-up section the energy is elevated from low to high. Our single criterion for segmenting a track into its sections is the amplitude.

Segmentation of music is a popular research topic. Many approaches have been made towards the automatic segmentation of generic or specific music genres [25, 40, 9, 22, 4]. Masataka Goto presents an audio player in which the standard seek controls are replaced with a 'jump to next section' function; the *SmartMusicKiosk*. Uninteresting parts such as intros can easily be skipped with this player. In addition to that, it can be used by the customers of record stores to quickly jump to the chorus of a track [11]. These approaches consider the for other music genres typical distinction factors such as timbre (the sonic characteristics), chroma and tempo, while we can restrict ourselves to only using the amplitude. Moreover, they aim at detecting repetitive sections and labeling them respectively. We simply need to label a section according to its average amplitude with bridge or chorus, independently from the other sections.

However, even if the MIR part in segmenting electronic dance music is easier than in other genres, these algorithms do not meet our requirements in two important points: Sections have to be aligned with the bars and, what is more, tracks shall be divided only into sections that measure a power of two. Today, accurately extracting the beat of a track no longer requires witchcraft [1, 19]. Obtaining sections that comply with the ordered structure of this music can however, not be guaranteed with the above methods. This is due to the last bars of a section, which might be varied slightly or drastically compared to the rest of the section (see paragraph 2.1). It might even be silent. Without knowledge about the composition of electronic dance music, a common segmentation algorithm might assign these last bars to the next section or even to a new section on its own.

Algorithm 1 shows the pseudo-code implementation of the segmentation algorithm that we propose. It is designed to respect the structural conditions as discussed in the following.

The minimum section size that can occur is one bar. Line 7 requests that amplitudes have to be similar. Firstly, this means that the amplitudes differ by not more than a threshold. A higher threshold might be used for the intro (i.e. as long as i is zero), to produce larger intro sections. Secondly, also if the current section constitutes a build-up (i.e. the maximum and average amplitudes are rising with every beat) new bars that fit to this trend have to be rated as similar. That is achieved by interpolating the average amplitudes and comparing the expected with the actual values. As long as the amplitudes are constantly rising, new bars have to be rated to be similar.

As we know the last bar in a section might be varied. Typically such variations consist of pausing one or several instrumental layers and thus the amplitude gets smaller (this is called a break). Therefore, line 9 tests, if the average amplitude of the current bar is greater than that of the current section so far. If so, the bar is not

Algorithm 1 Segmentation respecting the 2^n structure

Input: track t
Output: sections $s[]$, labeling $l[]$ (for all sections $l[] \in \{intro, bridge, chorus, outro\}$)

do beat extraction on t
$b[] \leftarrow$ divide t into 1 bar sized segments ▷ 1 bar = 4 beats
$s[0] \leftarrow$ create new section ▷ begin the segmentation
$i \leftarrow 0$
for $j \leftarrow 0$ **to** length$(b[])$ **do** ▷ for all 1 bar segments do
    $a[j] \leftarrow$ avg-Amplitude$(b[j])$ ▷ calculates mean value
    if ($a[j]$ and avg-Amplitude$(s[i])$ similar) **then**
        keep current section
    else if ($a[j] >$ avg-Amplitude$(s[i])$) **then**
        $i \leftarrow i + 1$
        $s[i] \leftarrow$ create new section
    else if ($\exists n \in \mathbb{N} : \text{size}(s[i]) = 2^n - 1 \wedge \text{size}(s[i]) \geq 3$) **then** ▷ size in bars
        keep current section
    else if ($\exists n \in \mathbb{N} : \text{size}(s[i]) = 2^n - 2 \wedge \text{size}(s[i]) \geq 6$) **then**
        keep current section
    else
        $i \leftarrow i + 1$
        $s[i] \leftarrow$ create new section
    end if
    $s[i] \leftarrow$ concat$(s[i], b[j])$ ▷ add bar segment to current section
end for
$l[0] \leftarrow intro$ ▷ begin the labeling
$l[i] \leftarrow outro$ ▷ i is the number of sections (counting from zero)
for $j \leftarrow 1$ **to** $i - 1$ **do**
    if (avg-Amplitude$(s[j]) <$ avg-Amplitude(t)) **then**
        $l[j] \leftarrow bridge$
    else
        $l[j] \leftarrow chorus$
    end if
end for
return $s[]$, $l[]$

considered to belong here and is put into a new section. If not, we know that the average amplitude of the section is bigger. Thus, it is likely that the bar is a variation at the section end and still belongs to the current section. Lines 12 and 14 decide whether such a varied bar, which is rated as 'not similar', might be added for the purpose of completing the section to a power of two. The first equality test in line 12 checks, if only one section is missing to complete a power of two. The second test checks, whether the section is big enough so that the rule can be applied. Thereby, the size of a section is defined to be the number of bars that already have been added to the array. The smallest sections that we allow to be completed with a not similar bar must have at least a size of four (the first variation can occur in bar four, so there will not exist smaller sections with varied ends). As small variations typically occur every four bars, these have to be captured by the similarity threshold. The exceptional rules should only take effect for the drastic variations at section ends. In some cases even two bars at the section end are varied. This scenario is addressed in line 14. Two bars long variations however, are only legitimate for section sizes greater or equal to eight, so that we do not accidentally interpret two small sections as a single one. If none of the cases are true, we add the bar to a new section, where it will belong to.

This algorithm runs in time linear to the length of the input track (i.e. it has the complexity of $\mathcal{O}(|track|)$). The main loop is repeated for each bar of the track. In every iteration average amplitudes and similarity tests have to be computed. The average amplitudes of sections can be calculated in constant time just as the average amplitude of a bar (e.g. by keeping variables that store the sum of the amplitudes and the number of bars for each section, updating them and calculating the mean value). The similarity test can also be done in constant time: Threshold testing trivially needs constant time and the interpolation can be approximated by only calculating the trend of the last two bars, which is legitimate here as the slopes are roughly linear. The existential quantifications can be written as finite disjunctions, as we limit the maximum section size to a reasonable value (e.g. 64 or 128 bars). Thus they can be checked in constant time.

From this point on we will assume that the segmentation of a track is given and that it is given as desired. That means that — if possible — the segmentation is fully consistent with the power of two sized sections out of which the track was composed. Furthermore, sections are assumed to be correctly labeled either with *intro*, *bridge*, *chorus* or *outro* and with a flag stating if they are vocal or not (i.e. if the section contains singing). Even if the algorithm could distinguish build-up and interlude sections by comparing the energy curves, both are still abstracted to be a bridge.

4.2 The Mixing Technique

The mixing method **natural mixing** differs from the other methods introduced as much as it resembles what human DJs are doing. It is geared to how traditional analog DJs mix, and thus, constitutes an interpretation of what ideal DJing should look like. Most techniques that automatic DJs are using (as listed in chapter 3, e.g. hpDJ and Mixtrax) have one thing in common: They first do the sequencing of tracks and on that basis they search for segues between successive tracks. However, quite often it is the case that if you consider two tracks you will not find a good or easy possibility to do a transition between them. Even if those tracks fit together, based on all thinkable musical characteristics, you might prefer to do a segue to a different track. A simple example would be two tracks in which all sections are vocal. As we do not want to play two vocal sections simultaneously, we cannot make a segue, even if the tracks otherwise fit perfectly. Our technique does not split the tasks of finding a sequencing and finding a segue into two different steps or algorithms. In one single search the sequencing and segues between all tracks are found, and thus, only results are generated that make no compromises. Besides, natural mixing is completely abstracted away from MIR. The last time the audio content of a track is looked at (before the actual playback) is in the segmentation step. In contrast, other approaches such as that of similarity matrices completely rely on MIR. We exclusively work with the model of a track's structure. This paragraph focuses on what musical qualities make sections and tracks fitting. How a mix compliant to those requirements is found is covered in chapter 5.

Doing a segue in natural mixing means playing two sections in parallel[1]. The whole technique focuses on which sections inside what tracks can when be played simultaneously. Other tasks, such as adjusting the tempos, equalizing or crossfading are also discussed, but they are considered secondary. The basic property which makes sections eligible to be played in parallel is their size. We want section borders to be aligned, so that the nice power of two structure of this music genre is preserved for the resulting mix. In addition to that, only by respecting the section boundaries we can be sure that the automatic DJ will respect the deeper structure (e.g. downbeats and varied bars) of the sections as it is not given lower level knowledge about their composition. Therefore, sections should be equally sized if they shall be played in parallel. But they do not generally have to be: If the level of abstraction would be lowered to musical phrases that are smaller than sections, then we could specify which of those can be combined. However, as this would increase the complexity of the problem we stay away from such freedom.

[1] Doing a segue by combining musically concluded parts (as we do it with sections) is often referred to as *phrasing*.

The pair of sections between which the segue is done has to be found within a pair of tracks that are musically compatible. We define tracks to be fitting, if their keys are in a harmonic relationship and if their tempos are compatible. Which key relationships are considered harmonic is discussed in paragraph 2.4. Hiromi Ishizaki et al. experimentally determined that a tempo alteration by less than 10% preserves the listener's comfort best [14]. We use their findings and define tempos to be compatible if the faster track does not differ by more than 10% from the slower one.

Within compatible tracks a pair of fitting sections can be chosen. Apart from being equally long, they also have to be musically compatible and respect the structure and energy curve of the mix. The latter means that for example after a build-up section in which the energy is increased, the audience expects a high energy section to follow. And thus, we should only allow high energy sections to be played after a build-up.

We propose the following range of possible kinds of transitions that strive to meet those requirements:

1. The intro section can be combined with any other section as it carries only little melodic content; if at all. The same holds for the outro. Therefore, the outro of the first track and the intro of the second track can be played in parallel. This constitutes the classical and unspectacular way to do a transition, which is sometimes also referred to as end-crossfading.

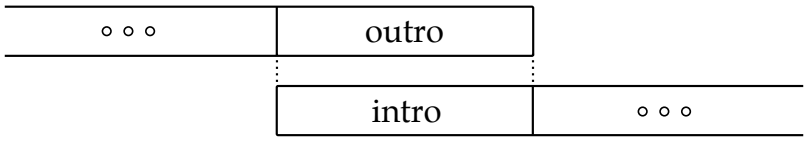


Figure 4.1: Outro to intro segues

2. Also the new track's intro can be started in the middle of the old track, parallel to a bridge or chorus section. Likewise the new track can be started with a bridge or chorus, parallel to the outro of the old track. Thus, at most one of the parallel played sections will contain rich melodic content.

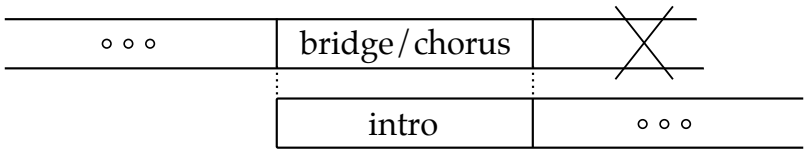


Figure 4.2: Bridge/chorus to intro segues

3. The next method is called the bridge swap: When one deck would play a bridge section in the next step, the other deck instead plays a bridge from a new track. The first track is stopped at the same time the new track is started, so in this kind of transitions tracks are not played in juxtaposition. Still the transition will appear seamless as the listener anticipates a bridge, yet without knowing what it will sound like.

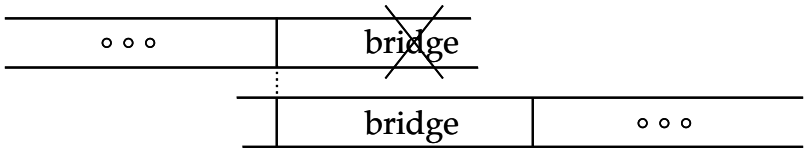


Figure 4.3: Bridge swaps

4. Alternatively bridges can be played in parallel. These kind of segues are especially prone to be disharmonic, as both bridges possibly are rich in melodies. Thus, the harmonic key relationship is of supreme importance for these kind of transitions. By that at least the melodies would harmonize.

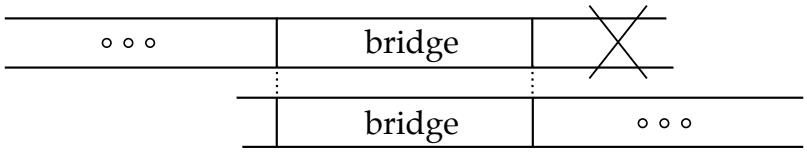


Figure 4.4: Superimposing bridges

5. Finally, we introduce a technique common in hip-hop DJing: Making cuts. Just as in the bridge swap both tracks are not playing overlapped, but the first one is stopped at the exact moment the next one is started. Cuts are done from the chorus or bridge of the first track to the chorus or bridge of the next track (in hip-hop choruses and bridges can often not be easily distinguished). This technique proves to be beneficial if both tracks are completely vocal or uninterruptedly contain melodies from the beginning to the end. This is often the case in hip-hop or pop music tracks which have not been especially edited for DJs.

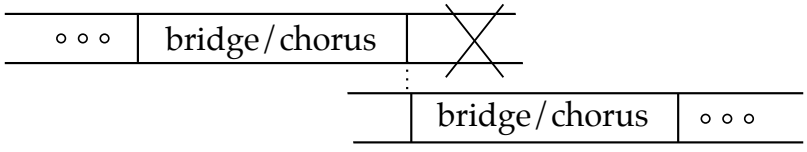


Figure 4.5: Cuts

Choruses are not played in juxtaposition as they have high amplitudes. Playing them together would require sophisticated equalizing to weaken frequency spectra in both tracks so that the resulting loudness would not stand out. Moreover, at no time two vocal sections should be played in parallel, as two singers singing simultaneously but independently will not harmonize. Besides, we require a transition to be at least eight bars long, to prevent too short segues that might be easily observable. We do not differentiate between constantly mid or low energy bridges and bridges were the energy is built up. This constitutes a source of error for the resulting energy curve of the mix. However, this is not due to some technical problems, but it is rather a design decision to risk minor flaws in the energy curve for the sake of simpler rules.

As a result, the mix preserves the mathematically nice power of two structure. The sections, between which segues have been done, now represent new sections that also measure a power of two bars. Furthermore, the mix will have a consistent energy curve (except for errors caused by false segmentation or by the too high abstraction of bridges). The intro section of the very first track will form the intro section of the mix and usually increase the energy to a mid level. During the course of the mix high and mid energies are altered repetitively: First, the energy is built up. Then, a high energy section follows, that is succeeded by a resolution leading to a mid energy regenerative section. Finally, in the outro of the last track, the energy curve is finished by bringing it down to a low level again.

Segues are smoothed by making use of the mixer. We assume the begin and end points of a segue to be given, as they are the points where the new track is started to play and the old one is stopped. Then, we interpolate the mixer controls between those points. In that way, the crossfader can be homogeneously moved from one extreme position to the other (e.g. from its leftmost to its rightmost position) during a segue. We can improve that movement by subdividing it into several stages. In fact it is harder to notice transitions in which the crossfading is not done monotonically throughout the whole segue. Namely by making the segue resemble one of the tracks own sections, in which additional instrumental layers are added. To achieve that, we will only adjust the crossfader in the first and last quarter of the overall transition time interval. In the first quarter the crossfader is homogeneously moved to its middle position, then in the following two quarters it is not adjusted and both tracks play equally loud. Finally, in the last quarter the crossfader is moved to its target position. Besides the crossfading, also equalizing is performed to make segues smoother and to avoid disharmonies between tracks. As this approach tries to exclude such cacophonic segues by construction, we do not care about detecting *clashing* frequencies and merely use the equalizers to smoothen the transition. This is achieved by using them to swap the bass lines of the tracks: The new track is started with the 'low' equalizer turned down (i.e. the bass frequencies are weakened) and

during the transition it is homogeneously turned up to the neutral position, while the old track's 'low' equalizer is turned in the opposite direction. Last but not least we need to discuss how tempo differences between two successive tracks are dealt with. As explained above, we do not allow tempo differences to be too large. So we are able to play both tracks at all tempos, that lie between their tempos, without causing discomfort; especially we can play each track at the other track's tempo without causing discomfort. As both, abrupt and continuous tempo changes quickly attract negative attention, we again adopt the method of human DJs: We try to make the tempo change as slow as possible. The new track is started at the old track's tempo and from that point on both tracks are played tempo synchronized. Their shared tempo is then linearly adjusted to the new track's tempo from the beginning to the end of the segue. The slower the tempo change is, the harder it can be detected. Thus, the longer a transition is, the better it is for adjusting the tempos. Furthermore, we can avoid changing tempos at all if they all oscillate around one tempo within the optimal range of ±10%. In that case each track is simply played at the average tempo of all tracks.

Finally, we require that a track is played at least long enough that one chorus section is played. Moreover, transitions should only be made from the second half of the first track to the first half of the new track. Otherwise a mishmash of tracks could evolve: Tracks could be started in their very last section or stopped in their first part directly after they were started.

It's still magic even if you know how it's done.

Terry Pratchett

Chapter 5

The Modeling Process

In this chapter we demonstrate how model checking is applied to the problem of synthesizing a DJ mix. Therefore, we create a formal model M of a DJ setup. In theory any number $n \in \mathbb{N}, n \geq 2$ of decks could be used. However, two decks suffice to generate an uninterrupted stream of music. This model is then implemented in SPIN's modeling language PROMELA and the desirable properties of a path through the model are encoded in a Linear Temporal Logic (LTL) formula φ. Then, M and φ are both given to SPIN, which searches an example path through the graph spanned by the model, that is in accordance with the properties described in the formula. The formal model constitutes an abstraction of a real life DJ setup, which is detailed enough to adequately simulate a DJ system but which is not too precise so that common computers could no longer cope with it. It offers all major controls that a DJ setup has, such as play, pause, loading a new track and seeking a position in the track. However, not all actions are reasonable at each point in time. Thus, we have to decide for each action when its availability should be restricted. Either it can be restricted by the model, depending on the current state, or it can be filtered out by the formula during the search. At the end of this chapter, we discuss how the model's speed was improved and how memory usage was minimized.

5.1 The Formal Model

In this paragraph we will formalize the model as Kripke structure. Kripke structures are used to describe reactive systems with infinite behavior. Even if we only consider finite mixes (as humans are likely to only spend finite time on listening to music) each mix can be described by an infinite path through the model: By appending an endless repetition of a state in which all decks are stopped to the finite mix. A Kripke structure is a tuple (S, I, R, L), where S is a set of states, $I \subseteq S$ is a set of initial

states, $R \subseteq S \times S$ is a set of transitions connecting pairs of states, and $L \subseteq S \times AP$ is a labeling with AP being a set of atomic propositions [5].

Each state of the model comprises of a set of variables, and therefore, our labeling L labels it with an assignment of these variables. A state is represented by a set of currently true atomic propositions. Hence, the set AP of atomic propositions is the set of all possible variable assignments.

Before we define the states of our model we introduce new data types, used to abbreviate an aggregation of other data types. The primitive data types used are *boolean*, *integer*, *string* and *arrays*.

$$\begin{aligned} \mathit{channel\_type} &= \mathit{integer} \times \mathit{integer}[] \\ \mathit{mixer\_type} &= \mathit{channel\_type} \times \mathit{channel\_type} \times \mathit{integer} \\ \mathit{flag\_type} &= \{\mathit{intro}, \mathit{bridge}, \mathit{chorus}, \mathit{outro}\} \\ \mathit{section\_type} &= \mathit{integer} \times \mathit{flag\_type} \times \mathit{integer} \times \mathit{boolean} \\ \mathit{track\_type} &= \mathit{section\_type}[] \times \mathit{integer} \times \mathit{integer} \times \mathit{string} \\ \mathit{deck\_type} &= \mathit{track\_type} \times \mathit{boolean} \times \mathit{integer} \end{aligned}$$

flag_type basically is an integer but the values 0 to 3 are addressed with *intro*, *bridge*, *chorus* and *outro*.

A state $s \in S$ of our model M is a tuple $(\mathit{deckA}, \mathit{mixer}, \mathit{deckB}, \mathit{time}, \mathit{track\_library}) \in \mathit{deck\_type} \times \mathit{mixer\_type} \times \mathit{deck\_type} \times \mathit{integer} \times \mathit{track\_type}[]$, where:

$$\begin{aligned} \mathit{mixer} &= (\mathit{channelA}, \mathit{channelB}, \mathit{crossfader}) && \in \mathit{mixer\_type} \\ \mathit{channelA} &= (\mathit{volumeA}, \mathit{eqsA}) && \in \mathit{channel\_type} \\ \mathit{channelB} &= (\mathit{volumeB}, \mathit{eqsB}) && \in \mathit{channel\_type} \\ \mathit{deckA} &= (\mathit{trackA}, \mathit{playingA}, \mathit{positionA}) && \in \mathit{deck\_type} \\ \mathit{deckB} &= (\mathit{trackB}, \mathit{playingB}, \mathit{positionB}) && \in \mathit{deck\_type} \end{aligned}$$

The mixer is represented by its two channels and a crossfader. Each channel has a volume fader and an array of equalizers. The crossfader, the volume faders and the equalizers are all abstracted to have a resolution that allows their complete value range to be described by integer numbers. A deck is represented by the track it is currently playing, a boolean variable that states whether this deck is playing or paused and the current playhead position inside the track (e.g. the number of bars already played). We are using discrete time (as we will justify soon), so we can actually save the current position and the time variable with integers (in the formal model, the value range of integers is not restricted).

The following definitions just exemplify the variables' structure. The names used are no actual variable names and we will not refer to them later.

$$\begin{aligned} section &= (id, flag, size, vocal) && \in section\_type \\ sections &= section[] && \in section\_type[] \\ track &= (sections, key, tempo, name) && \in track\_type \end{aligned}$$

Each track that might be played in a deck stems from the track library, which saves information about each track's segmentation, keys and tempo. If we want to make a mix out of $n \in \mathbb{N}$ tracks, the track library is defined to consist exactly out of those n tracks. The single tracks of *track_library* are accessed with $track[0]$, $track[1]$, ..., $track[n]$. For all $j \in \{0, ..., n\}$ it holds that $track[j] \in track\_type$. Thereof $track[0] = ([], 1, 1, \text{"}empty\text{"})$ is a dummy track that is loaded in a deck instead of saying the deck is empty; it represents the 'empty track'. The camelot keys are mapped to integer numbers as follows: $1A \rightarrow 1$, ... , $12A \rightarrow 12$, $1B \rightarrow 13$, ... , $12B \rightarrow 24$. The tempo is given in BPM. Moreover, each track carries a unique name, so that it can be identified by the human user and distinguished by computers. The segmentation consists of an array of sections. Each section therein is assigned an id which subsequently numbers the sections of the track, its type, its size in bars and whether it is vocal or not. The section types are represented by integer numbers that stand for intro, bridge, chorus or outro.

Except for *time*, all variables in M have finite value ranges, or can be restricted to be finite: The rotary and linear potentiometer resolutions are abstracted to be incremental (as it is also done in digital DJing systems). And to suit the reality, we assume that tracks are always finitely long, having finite tempos and names. As a consequence they will too have only a limited number of finitely sized sections.

The set of initial states I consists of the single state s_0. It represents the default start state of a DJ setup: $s_0 = (deckA_0,\ mixer_0,\ deckB_0,\ 0,\ track\_library)$. Indices are used to distinguish different states and their variables. By convention, we will play the first track of the mix in *deckA* so the crossfader in $mixer_0$ is at its leftmost position and the equalizer controls are set to their neutral default position. We define the initial states of the decks to be: $deckA_0 = deckB_0 = (track[0],\ false,\ 0)$.

An indexed state is defined as follows:

$$s_i = (deckA_i,\ mixer_i,\ deckB_i,\ time_i,\ track\_library)$$

The index $i \in \mathbb{N}$ is propagated through the variables of the state. For example $deckA_i$ looks like $(trackA_i,\ playingA_i,\ positionA_i)$. If we are not interested in the whole state of s_i, we will just itemize the state's indexed variables that we look at.

Pairs of states $(s_1,\ s_2) \in S \times S$ are in the set of allowed transitions R if and only if they are in one of the following relations:

- $R_{timestep} = \{(s_1,\ s_2) \in S \times S \mid \forall\, deck \in \{deckA,\ deckB\} : (playing_1 \wedge playing_2 \wedge position_2 = position_1 + 1) \vee (\neg playing_1 \wedge \neg playing_2 \wedge position_2 = position_1) \wedge time_2 = time_1 + 1\}$
  The disjunction has to be separately valid for both decks. $deck \in deck\_type$ is defined to be the tuple $(track,\ playing,\ position)$. Thus, *deck* can take both, the value for *deckA* and *deckB*. The increase in the playing position for each time step is assumed to be one. A pair of states is in this relation, if time progressed by one step and if the positions were also increased adequately for playing decks.

- $R_{load} = \{(s_1,\ s_2) \in S \times S \mid \exists\, deck \in \{deckA,\ deckB\} : track_1 \neq track_2 \wedge \neg playing_1 \wedge \neg playing_2\}$
  A pair of states is in this relation, if a new track was loaded into a deck (i.e. the deck's track is replaced by a different one). The deck has to be paused when the track is changed.

- $R_{play} = \{(s_1,\ s_2) \in S \times S \mid \exists\, deck \in \{deckA,\ deckB\} : playing_1 \neq playing_2 \wedge track_1 = track_2 \neq track[0]\}$
  If a deck changed its playing state, a pair of states is in this relation. That means either it was started or stopped. Therefore, a track needs to be loaded; the dummy track $track[0]$ is not allowed to be played.

- $R_{mixer} = \{(s_1,\ s_2) \in S \times S \mid crossfader_2 = crossfader_1 \pm 1 \vee (\exists\, eq \in eqsA \cup eqsB : eq_2 = eq_1 \pm 1) \vee (\exists\, volume \in \{volumeA,\ volumeB\} : volume_2 = volume_1 \pm 1)\}$
  If a mixer control was changed, the states are in this relation. To obtain an arbitrary change in an equalizer or fader value, the increment operation quantized by one has to be performed repeatedly. The set union operator "$\cup$" is intuitively used to combine the two arrays of equalizers; same goes for the operator "$\in$" that we here use to state that the element has to be one of the array's elements.

Let $R = \{(s_1,\ s_2) \in S \times S \mid (s_1,\ s_2) \in R_{timestep} \vee (s_1,\ s_2) \in R_{load} \vee (s_1,\ s_2) \in R_{play} \vee (s_1,\ s_2) \in R_{mixer}\}$. Loading tracks, changing the playing state and modifying the mixer controls are not required to consume time. Thus, within each timestep infinitely many such actions can be performed. We therefore say, that they are performed in between timesteps and that the time point after the last performed timestep is also the execution time of the actions.

Now, a *path* in a Kripke structure M is an infinite sequence $\sigma = s_0,\ s_1,\ s_2,\ ...$ such that $s_0 \in I$ and successive states $(s_i,\ s_{i+1}) \in R$. Let *Paths* be the set of all such

sequences and $n \in \mathbb{N}$ be the number of tracks in the track library. We define a **mix** to be a path that plays each track of the library. The set of all mixes for the model is the set:

$$\{s_0,\ s_1,\ s_2,\ \ldots \in Paths \mid (\exists(i_1,\ \ldots,\ i_n) \in \mathbb{N} \times \ldots \times \mathbb{N}\ \forall j \in \{1,\ \ldots,\ n\}\ \exists\, deck \in \{deckA,\ deckB\} : track[j] = track_{i_j} \wedge playing_{i_j}) \wedge (\exists k \in \mathbb{N}\ \forall l > k\ \forall\, deck \in \{deckA,\ deckB\} : \neg playing_l)\}$$

$track[j]$ is a track of the library and $track_{i_j}$ and $playing_{i_j}$ belong to the variable *deck*. So a path is a mix, if a set of n time indices exists such that there exists a state for every track of the library, in which the track is loaded in a deck while the deck is playing, and if after a certain time index both decks stay paused forever.

Why do we use discrete time in our model? We can do that without loosing freedom in movement as we are only interested in the parts of the execution, where an action is carried out (i.e. play, pause or load) or a section boundary of a track is crossed. We play two tracks tempo-, beat-, downbeat- and even section-aligned and thus, the play action cannot be executed at any point in dense time. Using discrete time steps dramatically reduces the number of possible points to start playing without reducing the capability of the system. Depending on the implementation of time steps, we exclude cases where tracks are not downbeat matched or even all cases in which they are not section aligned. Theoretically tracks could be loaded at any point in dense time, but as these actions cannot be heard, it does no harm to round them down to the next smaller point in discrete time. We only really have to increment the position if that is of use for the play/pause action, and thus, for the correct alignment of parallel playing. Between the start and stop points of a track the playback continues normally.

This model is now capable of simulating two players that play tracks on their own or synchronized in parallel. However, not every mix that can be made will actually sound good. Therefore, for every state we limit the transitions that can be made. We achieve this by proposing a temporal logical formula that has to be true in every state of M; this will make several states illegal and thus, the transitions to those states may not be taken. This formula encodes the rules and techniques from chapter 2 and 4.2. Paragraph 5.3 discusses the creation of such a formula. In the following we will call the paths/mixes that can be done in the model **legitimate** and those that also fulfill the temporal logical formula (i.e. the good sounding mixes) the **desirable** ones.

5.2 The Spin Model

To be able to use this model in SPIN we need to translate it into a computer program, implemented in the language PROMELA. The states of this program (i.e. the sets of all its variable values) should be identical with those of the model. In general, during the execution of a program, the variables are manipulated in a way such that a target is achieved (e.g. a certain variable holds a specific value). Our program will perform such manipulations too, change the variable states and by that changes its state: In a way such that each successive pair of states is contained in the transition relation R. The target of this program is not to compute some dedicated values using an algorithm, but it rather is to simulate a DJ system in the same way M does. Each state a DJ system might take is abstracted to be a state of the program. Moreover, the states of the program are reached in the same way they are reached in the DJ system. So a state of the program can for example not be set to 'both decks are playing a track in section two' in one single step, but this state has to be reached through a sequence of actions: Tracks are loaded to the decks, then the decks are started and timesteps are carried out to reach section two.

Our PROMELA program will not be a pure implementation of the model, but it will be extended in several points. For example the state changes that are defined above cannot always be translated to a single state change of the program. They have to be simulated by a sequence of variable manipulations, as explained in paragraph 5.2.2. Then, for reasons of efficiency, we will not encode all DJing rules and techniques into the temporal logical formula. Some will be directly included into the program as constraints that limit the allowed transitions outgoing from each state. As a result, the search graph will be reduced. Furthermore, the implementation of the formula required some modifications of the program as described in 5.3. Last but not least, paragraph 5.5 discusses tweaks on the model that improve execution speed and minimize memory usage of the model checking process.

5.2.1 Promela

PROMELA (short for Process Meta Language) is a powerful modeling language to describe distributed systems and communication protocols. Yet, as model checking of multi process systems generally is a harder task than that of single process systems, we will implement our model using one process only. A PROMELA program consists of processes, variables and message channels. Merely the parts of the C-like syntax that are used in the following are introduced below.

Variables can be defined global, or local for a process, and they can only be modified by processes. `proctype` declares an executable process including its behavior. It

might be declared `active`, too; then it is executed from the initial system state on.

```
active proctype pname()
{ statements }
```

The standard datatypes `bit`, `bool`, `byte`, `short` and `int` are available for variables. They can be used in arrays, too.

```
bool bname = true;
int iarray[] = 0
```

`mtype` is used to introduce new symbolic constants. First, `mtype` has to be initialized, then new variables of this type can be declared and normally used.

```
mtype = {Playing, Paused};
mtype deckState = Playing
```

If more complex data structures are required, they can be defined using a `typedef`. They are accessed in a C-like manner.

```
typedef tname { bool b; int iarray[2] };
tname foo;
foo.iarray[1] = 3
```

Note that the semicolon ";" is used to *separate* statements. With the `atomic` construct several execution steps of the program can be combined into a single state change of the system. `d_step` works in the same way as `atomic`, however, it does not allow for nondeterminism and can be executed more efficiently.

```
atomic { statement; ... ; statement }
```

Nondeterminism becomes possible with options, which start with a "::". Then, a sequence of statements follows, whereof the first statement is called a guard and has to return a boolean value. The guard is not separated with a semicolon bot with an arrow "->". Alone when it evaluates to true, the option can be taken and the whole sequence is executed. Case selection (`if`) and repetition (`do`) constructs are used for implementing the control flow. In case selections one option — for that the guard evaluates to true — out of several possible options is chosen nondeterministically and executed. In the repetition construct this is done repetitively.

```
if
        :: (true) -> foo.iarray[1] = 4
        :: (deckState == Playing) -> deckState = Paused
fi;
do
        :: (true) -> foo.b = true
        :: (true) -> foo.b = false
od
```

For a complete description of PROMELA's capabilities please refer to [12].

5.2.2 Implementation in Promela

In this paragraph, the essence of the formal model's implementation in SPIN is presented. The development turned out to be challenging as we are using SPIN the other way round: If the model is incorrect no mix can be found. Hence, SPIN merely returns a 'satisfied', which gives no further knowledge about where the errors in the model are. If the model is correct a counterexample is returned, if it is only partly correct a counterexample will be returned but will not represent a valid mix.

We begin with defining a range of assumptions:

- Tracks are realistic and conform with fundamental composition conventions: They have finite length, exactly one intro and outro, and at least one chorus section. Further, they need to have a constant tempo and unambiguous key. The number of sections and each section size are restricted to 255, so that they fit in a byte typed variable. The overall number of bars of a track has to fit in a short data type, the tempo has to be between one and 255 and the number of tracks to be mixed is restricted to 255. Especially mixes then will be of finite duration. Furthermore, intros should not exceed the first half of a track and outros should not start earlier than in the second half.
- The set of tracks given has to provide possibilities for creating a mix according to our rules. For example if a harmonic mix has to be computed but the tracks can just not be ordered to be all in harmonic relationships, SPIN will simply output that it did not find a mix.
- The program is assumed to be executed on a computer with infinite physical memory. Otherwise, SPIN will cancel its execution if the limits are exceeded.

Next, we will demonstrate how a state of model M is realized in the program. Much like in M, the set of all variables form a state of the program. Exactly the tracks to be mixed are embedded in the program. Each track is represented by the following data structure, which is derived from a track's definition in paragraph 5.1.

```
1  mtype = { Intro, Chorus, Bridge, Outro, Empty };   /* section types */
2  typedef track {
3          byte id;     /* unique for every track (replaces track name) */
4          byte key;
5          byte BPM;
6          short bars;
7          byte nrSections;           /* -> maxNrSections = nrSections */
8          mtype section[maxNrSections];                    /* the flag */
9          byte sectionSize[maxNrSections];            /* size in bars */
10         bool vocal[maxNrSections] = false
11 }
```

The id of a section is its array index, starting from one. `section[0]` is assigned the section type `Empty`. Likewise, decks are implemented in a straightforward way. `position` is the id of the current section played, `flag` and `flagSize` are set accordingly.

```
typedef deck {
        bit id;                 /* 0 = deck A, 1 = deck B */
        track t;
        byte position;
        bool playing;
        mtype flag = Empty;     /* the flag of the current section */
        byte flagSize
}
```

So far, we abide by the formal model M. However, we will not implement the mixer section. Calculating the mixer settings in every step does considerably consume processor time; and while the process of searching a path through the model, they would be computed in vain for all bad trials. Luckily we can derive the mixer state from the deck states. Given the start and end points of the segue they can be calculated in a postprocessing step (independently of the PROMELA program) as explained in paragraph 4.2.

We have seen what a state of the program looks like, but how is it manipulated? In an initialization step, the tracks are declared and initialized, as well as two decks `A` and `B` are created. Then, the main loop is started, simulating the DJ system. `step(deck)`, `fits(deck, track)` and `load(deck, track)` are inline code calls, which means they are just placeholders for more code (much like preprocessor macros). The main loop essentially looks like this:

```
do
        /* do a timestep and play next part of the tracks */
        :: atomic { step(A); step(B) }

        /* load new track */
        :: atomic {
                if
                :: fits(A, track_x)     -> load(A, track_x)
                :: fits(B, track_x)     -> load(B, track_x)
                :: ... /* same pairs of options for all other tracks
                    */
                fi
                }
od
```

Either a timestep is done with `step(deck)`, for each deck subsequently, or a new track is loaded, if it fits. This means, that it has to fit in tempo and key with the track currently playing and the deck it is to be loaded to has to be stopped.

Furthermore, it is tested if the track was played before; tracks are only allowed to be played one time. A boolean array keeps track of the 'already-played' statistics for the tracks. The loading options are grouped using `atomic` and not `d_step`, as the track to load should be chosen nondeterministically. The same goes for executing timesteps. Several options are possible: Play and pause actions for a deck are done in combination with the timestep action, to indicate when the playing status of a deck changed. If the track was playing before, it can be continued to play or it can be stopped after the timestep. Likewise, if the track was stopped it can stay stopped or it can start playing in this timestep and launch the play action therewith.

With using `fits(deck, track)`, we deviate from the model, as we include constraints to the program that originally should have been included in the specification. In the original model any track could be loaded. Here we limit the possible state changes by allowing to load fitting tracks only, and thus, constrain the program. As a result, the model checking will run through faster as the program has less possible transitions. Paragraph 5.3 again picks up the topic, whether undesirable transitions should be disallowed by program constraints or by the specification.

`load(deck, track)` copies all fields of the new track to the specified deck, as we cannot simply duplicate a variable reference in PROMELA. The track can be accessed via `deck.t`. Now, a section within some track's first half is chosen to be the section in which the playback is started. `deck.flag`, `deck.flagSize` and `deck.position` are set accordingly. All of the execution steps needed to achieve the loading are regarded as a single step, as they are embraced by an `atomic` block.

So, by what amount is the position in a track incremented in each timestep done by `step(deck)`? Two different step sizes appear constructive:

- **Bar sized steps.** They are our simplest and canonical alternative to simulate the time progress in discrete time, as actions can only be done aligned with the bars. Within a bar, tracks are not permitted to be started or stopped, as they are not allowed to be played in parallel if their bars are not aligned. Favorable is, that one bar sized steps allow the model to simulate every bar synchronized behavior, and thus, every possible reasonable behavior in reality. But this characteristic also poses a drawback. The program has a lot of freedom of what to do next. Thus, the number of possible states becomes huge. As we want to further align sections, we would appreciate a time model that abstracts away the possibility to do an action within a section.
- **Section sized steps.** Increasing the position by the size of the currently playing section achieves exactly that. Now, the model is exempt from states that represent not section aligned playing of two tracks. Tracks can only be started or stopped at section borders. The size by that the positions are incremented

changes dynamically. If only one deck is playing, the step size is the size of the currently playing section (saved in variable `deck.flagSize`). If both decks are playing, they will be playing an equally sized section (as ensured by the specification in paragraph 5.3), and thus, any of the decks' section sizes can be taken as step size. Section sized steps accurately model the behavior that we need.

In the following, we will always use the section sized steps model.

What actually is done in `step(deck)` depends on what option is chosen nondeterministically. If the track was stopped it can stay stopped and nothing is done; or it can be started. Decks play tracks in a strictly alternating way. So it has to be this deck's turn to start a track. In addition, a track has to be loaded to the deck and the other track should be playing in its second half. If all that holds, a stopped deck can be started. If the deck was playing before, it can simply continue to play. Thereby, the track is not allowed to be played in its last section; in this case it can only be stopped. A playing track can be stopped anywhere in its second half. After each increment of position `deck.flag` and `deck.flagSize` have to be updated. Also it is checked if the chorus of the current track was played. If so, the whole track is marked as played and if every track is marked as played the program is signaled to stop the playback.

We also add a few useful features to the PROMELA implementation that have not been included in the formal model:

- The program can be parameterized. The maximum tempo difference between subsequent tracks can be set as desired.
- Also we add the possibility to restrict the play-length of each track. The system keeps a record on how many bars of a track have been played. The playback might only be stopped if the number of bars played is within a specified range.
- Other changes (as previously mentioned above) include: A new track is tested if it fits before it can be loaded, the check if all tracks have been played, keeping a record of which tracks have been played, and that decks are forced to strictly alternate.
- In the bar sized steps model unequally sized sections can be played simultaneously together. This is a useful feature, as we might want to make a segue by playing the last section of the first track along with a larger section of the new track. To demonstrate that we do not completely loose this capability in the section sized steps model, we add the possibility to split a section into two parts when the track is started. Those new sections can then be aligned with

the first track. The split is implemented by two new options in `step(deck)`. One that starts the track and splits the current section and another one that plays the second part of the split section.

- `step(A)` and `step(B)` should actually be performed at the same time. Yet, this is not possible using one process only. Problems arise as both decks access each others playing state. If the first deck is stopped after the current section was played, it would appear as 'stopped' to the second deck. However, it was playing during the time step. The second deck just sees its state after the execution of the step. Thus, instead of stopping a deck a marker is set. If a marker is set, it has to be handled in the main loop before any other option can be executed. The handling routine then actually stops the deck and sets its state accordingly. This trick simulates the simultaneous execution of both decks' steps; hence it enables us to renounce a model with multiple processes.

But how do we obtain a mix from an execution path through this program? The state graph (i.e. all possible states of the program and a state is connected to another one if, and only if, it can be reached within one step in the program) tries to emulate the formal model. But we want to avoid analyzing sequences of states through the state graph in order to create a sequence of snapshots of the DJ setup. It is much easier to extract a sequence of actions that, when applied to a DJ setup, provides us with the mix. To create this sequence, the program in chronological order prints out each action it performs, along with the respective position and data about the tracks' sizes and flags after each time step. If track y was loaded in deck X, `'deck_X loaded - track_y'` is printed. After each time step a pair of data about both decks is printed. It consists of whether the deck was started or stopped, of the current position and of the size of the current section: `'start/stop, position=current_section_id, size=current_section_size'`. Figure 5.1 shows the printouts of an exemplary counterexample path, which represent the automatically synthesized mix resulting from the model checking run.

5.3 The Specification

We are given the PROMELA model explained above and want to find a path in it, that adheres to our mixing technique. Thus, we will define a temporal logical formula φ that will serve as specification for the model checking runs. SPIN comprehends specifications written in LTL, which we will introduce first. Then, we discuss how DJing rules are encoded in LTL.

```
deck A:                                        deck B:
A loaded - LifeGoesOn
start     position=1, size=64   Intro    -           paused
          position=2, size=8    Bridge   -           paused
          position=3, size=16   Bridge   -           paused
          position=4, size=8    Bridge   -           paused
          position=5, size=16   Chorus   -           paused
          position=6, size=16   Chorus   -           paused
          position=7, size=4    Chorus   -           paused
          position=8, size=16   Chorus   -           paused
B loaded - Riverside
stop      position=9, size=32   Chorus   -  start    position=1, size=32   Intro
          paused                         -           position=2, size=8    Bridge
          paused                         -           position=3, size=16   Chorus
          paused                         -           position=4, size=16   Chorus
          paused                         -           position=5, size=16   Bridge
          ...                            -  ...
```

Figure 5.1: Printouts of an exemplary counterexample path

5.3.1 Linear Temporal Logic

Let AP be a set of atomic propositions as defined in paragraph 5.1, let $a \in AP$. The following grammar defines the **syntax** of LTL:

$$\varphi ::= a \mid \neg\varphi \mid \varphi \vee \varphi \mid \mathbf{X}\,\varphi \mid \varphi\,\mathbf{U}\,\varphi$$

X is called the *next* and **U** the *until* operator.
Let $\sigma \in \mathbb{N} \to 2^{AP}$ be an infinite word. Infinite words can be understood to be an assignment of the set of true atomic propositions (i.e. the current variable values) to every state in an infinite path/mix. The **semantics** are defined over infinite words σ as follows:

$$\begin{aligned}
\sigma, i &\models a && :\Leftrightarrow a \in \sigma(i)\\
\sigma, i &\models \neg\varphi && :\Leftrightarrow \sigma, i \not\models \varphi\\
\sigma, i &\models \varphi \vee \psi && :\Leftrightarrow \sigma, i \models \varphi \text{ or } \sigma, i \models \psi\\
\sigma, i &\models \mathrm{X}\,\varphi && :\Leftrightarrow \sigma, i+1 \models \varphi\\
\sigma, i &\models \varphi\,\mathrm{U}\,\psi && :\Leftrightarrow \exists n \geq i : \sigma, n \models \psi \text{ and } \forall m \in \{i, ..., n-1\} : \sigma, m \models \varphi
\end{aligned}$$

with $i \in \mathbb{N}$, $a \in AP$ and φ, ψ being LTL formulas. Then, $\sigma \models \varphi$ holds if, and only if, $\sigma, 0 \models \varphi$. Let M be a Kripke structure and let *Words* be the set of all infinite words in this model, equivalently to the set *Paths*. The $\models$ operator is defined on Kripke structures, such that:

$$M \models \varphi :\Leftrightarrow \forall \sigma \in \mathit{Words} : \sigma \models \varphi$$

Rule	as constraint	in LTL
limit tempo difference	✓	
align section beginnings	✓	
harmonic mixing	✓	✓
playing all tracks	✓	✓
limit playtime per track	✓	✓
cuts or overlapping segues	✓	✓
sequencing (order)		✓
segue duration		✓
segue techniques		✓

Table 5.1: Overview about how DJing rules are realized

We use the additional operators *finally* (**F**) and *generally* (**G**) that are defined as follows:

$$\mathbf{F}\ \varphi \equiv \mathit{true}\ \mathbf{U}\ \varphi$$
$$\mathbf{G}\ \varphi \equiv \neg(\mathbf{F}\ \neg\varphi)$$

The LTL formulas that are stated in the following all describe so called **safety properties**. A LTL property is a subset T of the set *Words* of all infinite words. Now, safety properties are those properties T, for that all infinite violating paths have a *bad prefix*: A finitely long path that violates the specification regardless of what infinite path follows. For all $\sigma \in \mathit{Words} \setminus T$ (so the violating words) it holds, that there exists a finite prefix $\hat{\sigma}$ of σ such that for all $\sigma' \in \mathit{Words}$ the concatenation of $\hat{\sigma}$ and σ' is not an element of T [3, pp. 111-115].

5.3.2 Encoding the DJing Techniques

We have two instruments at our disposal to restrict the states that the model can take, and thus, to limit the number of allowed transitions: A constraint can be added to the program or we can encode the restriction into the LTL specification. Table 5.1 gives an overview on how we realized each rule and restriction. In the following, we explain how the rules that we chose to realize in LTL have been encoded.

Supporting harmonic mixing was accomplished by a combination of using constraints and a LTL formula. It could not be encoded completely in LTL, as not every set of tracks can be ordered to be completely harmonic (e.g. the set of two tracks with keys 5A and 10A). Therefore, we sought for a mechanism that tries to order tracks as harmonic as possible; that means it should maximize the overall number of harmonic segues. In LTL only definite statements can be made: 'Every segue has to be harmonic' or 'No segue may be harmonic'. The resolution for this problem was

to let the program keep a record on how many harmonic transitions it did, and then, to request a minimum percentage of harmonic transitions in the formula. Possibly, the model checking process can be done repeatedly, while increasing the threshold on the minimum percentage until no further a counterexample can be generated. The formula we use looks like:

$$\mathbf{F}\ (\mathbf{G}\ (\mathit{harmonicValue} \geq \mathit{threshold}))$$

As explained in paragraph 5.2.2, the program also keeps a record of which tracks have been played and whether all tracks have been played (saved in the boolean variable $\mathit{played}[0]$). The next formula requires that at some point in time all tracks should have been played, and from that time on both decks have to be paused and always stay paused. This ensures that we merely obtain finite mixes. A and B are the two decks.

$$\mathbf{F}\ (\mathbf{G}\ (\mathit{played}[0] \wedge \neg \mathit{A.playing} \wedge \neg \mathit{B.playing}))$$

Once the playback has started, no state may exist in which both decks are paused at the same time before the mix is finished. The boolean variable *onAir* is set to *true* when the playback is started first.

$$(\mathit{onAir} \rightarrow (\mathit{A.playing} \vee \mathit{B.playing}))\ \mathbf{U}\ \mathit{played}[0] \tag{5.1}$$

This formula actually disallows the technique of doing cut segues, as during a cut there exist states (just in the program) in which both decks are paused. Therefore, we use this formula to force a mix to purely consist of overlapping transitions.

We also have to add a little fix to hinder the decks from playing the empty track, or from playing a different section than an intro when the mix is started.

$$(\neg \mathit{A.playing} \wedge \neg \mathit{B.playing})\ \mathbf{U}$$
$$((\mathit{A.flag} = \mathit{Intro} \wedge \mathit{A.playing}) \vee (\mathit{B.flag} = \mathit{Intro} \wedge \mathit{B.playing}))$$

Now, that the tracks are being played, we can limit the time that each track is played (as mentioned in paragraph 5.2.2). The model counts the bars that have been played in each track and the specification requires this number to always be within a certain range. The lower boundary is not checked by the formula, as the model is constrained to only be able to stop a deck after the boundary has been exceeded.

$$\mathbf{G}\ (\mathit{A.barsPlayed} < \mathit{maxBarsPlayed} \wedge \mathit{B.barsPlayed} < \mathit{maxBarsPlayed})$$

Moreover, we have to make sure that no two vocal sections are played simultaneously:

$$\mathbf{G}\ (\neg A.t.vocal[A.position] \vee \neg B.t.vocal[B.position])$$

When it comes to making a segue, the program has many choices on how to do it. First of all, it can choose between making a cut or an overlapping segue. When two tracks can not be played in parallel, often a cut can be made nonetheless. Unfortunately, the program also does cuts when an overlapping segue is possible. As we have the same problem that we had with implementing harmonic mixing, we chose the same solution: If no mix using merely parallel segues was found, we can set a minimum percentage of segues that have to be played in parallel.

The specification controls and powers the traversal through the program's state graph. While the constraints partly dictate what segues are undesirable, SPIN will choose one of the nondeterministically modeled options according to the specification. The sequencing is (in theory) chosen nondeterministically within the limits of the constraints and the harmonic mixing formula. But how are segues chosen?

The musical alignment is to a big extent determined by the model, however, the segue durations and what sections can be combined are still undetermined. Thus, we have to encode the transition types of page 37. These will configure what sections can be played together and when a deck is stopped. Therewith, they set the segue duration, which always tries to last as long as possible. If the sections of two tracks cannot be aligned any further, than the segue is completed. However, the segue techniques rely on a notion of time and partly need to talk about future or past: The section that was played before the current one or that will be played next. Encoding these segues with LTL's **X** operator failed, as it demands somewhat more runtime than a formula does without it. Luckily, the model does not allow the deck to wildly alternate the playing of different sections of a track, but it predictably plays one section after the other. Therefore, we can add two variables *suc* and *pre* to the model that contain the successor and predecessor section instead of using the **X** operator. By that, the formula's demands of knowledge about the future and past of a track's playback are met. Type 1., 2. and 4. for doing a segue are encoded in the following manner:

$$\begin{aligned}\mathbf{G}\ &(A.playing \wedge B.playing)\\ &\rightarrow\ (\quad (B.flag = Outro\ \wedge\ A.flag = Intro)\\ &\qquad \vee (A.flag = Outro\ \wedge\ B.flag = Intro)\\ &\qquad \vee (B.flag = Outro\ \wedge\ A.flag = Bridge)\\ &\qquad \vee (A.flag = Outro\ \wedge\ B.flag = Bridge)\\ &\qquad \ldots\ )\end{aligned}$$

If both decks are playing, the sections that they are playing have to form pairs as specified. In addition to the pairs shown, also a chorus and intro or two bridges can be played in parallel. However, in these two cases the chorus of the old track is required to have been played at least once, so that the track is given the chance to develop itself (this is encoded as e.g. $A.flag = Chorus \wedge B.flag = Intro \wedge played[A.t.id]$). Last but not least, we consider the encoding of transition type 5. which includes 3. implicitly. In contrast to the other formulas, this one is written down in a negative way. It describes what cuts we do not wish. The following formula is just one case; it needs to be duplicated with deck A and B swapped to work in both ways.

$$\begin{aligned}\mathbf{G}\ (&A.playing \wedge \neg B.playing \wedge B.t.id \neq 0 \wedge \neg played[0]) \\ &\rightarrow\ (\quad A.flag \neq Intro \\ &\qquad \wedge B.pre \neq Outro \\ &\qquad \wedge A.flag = Chorus \rightarrow B.pre = Chorus \\ &\qquad)\end{aligned}$$

In this case the cut was made from B to A. If only one deck is playing and the other one is not uninitialized (i.e. A did not just start the mix but actually B was playing before A), as well as the mix is not being finished and thus no more tracks should be started, then the following holds: The newly started deck A should not play an intro section, as we do not want to make cuts into intros. Moreover, deck B should not have played an outro before; we also do not want to make cuts from outros. Last but not least, if A is playing a chorus section then the last section that was played by B has to be a chorus too. As a consequence, cuts can be made from chorus to chorus, from chorus to bridge or from bridge to bridge. They cannot be made from an intro or into an outro due to constraints in the program: Tracks can exclusively be stopped in their second half (which does not contain any intros), and they can only be started within their first half (which does not contain any outros). Transition type 3. (the bridge swap) is implicitly included in this formula, as the formula also enables cuts from the chorus or bridge of the first track to the bridge of the next one. This includes those cases, where the chorus or bridge of the first track was succeeded by a bridge.

Earlier we proposed two different methods to implement time steps. Bar and section sized steps differ in the LTL formula in one point: Bar sized steps still needs to be further aligned.

$$\mathbf{G}\ (A.flagSize = B.flagSize \rightarrow A.offset = B.offset)$$
$$\mathbf{G}\ (A.playing \wedge B.playing \wedge A.flagSize \neq B.flagSize) \rightarrow (A.offset = 0 \leftrightarrow B.offset = 0)$$

In the bar step model an *offset* variable is added to both decks. It counts the bars played in each section. The first claim states that if the section sizes are the same, then the sections have to be played in synchronization. If the section sizes are unequal and nonetheless both decks play, then the beginnings of the sections have to be aligned and the rest will stay aligned as long as both sections are being played (on page 51 we implemented this exception rule for section sized steps, too).

All of the formulas explained above have to be valid. Thus, we combine them with the logical operator "$\wedge$". The overall specification φ is then obtained by negating this conjunction.

5.4 Obtaining a Mix

The model checking tool SPIN has to decide whether M satisfies φ, so it checks whether φ holds in every state of the model M. It accomplishes this by trying to violate the specification (i.e by searching a counterexample path in which the specification is violated): It negates φ and searches a path through M that fulfills $\neg\varphi$ (an example for $\neg\varphi$ poses a counterexample). In our case, a double negation is obtained, which is canceled out and thus, SPIN searches for an example that fulfills the conjunction of the formulas above; it actively searches a mix. An optimized nested-depth-first graph search algorithm is used for that. More information about SPIN's technical details can be found in [13]. Breadth-first search can also be used to find the shortest mix, however, it consumes too much memory to be practicable. If we want to find the shortest mix we instead use SPIN's iterative deepening depth-first-search algorithm.

The model and specification are both saved to the file `model.pml`. SPIN does not run the search itself. It creates a C program — called PAN — that runs the verification. '`spin -a model.pml`' checks the model's syntax and then creates this C program. We need to compile it using '`cc -o pan pan.c`', while possibly adding compilation parameters that e.g. exchange the search algorithm, randomize the search or limit memory use. '`./pan -a`' finally carries out the verification; the directive '`-a`' signals PAN that it has to check a safety property and that it has to search for an acceptance cycle (a path followed by a cycle that fulfills $\neg\varphi$). If a counterexample was found, it is saved in raw form to `model.pml.trail`. To be able to read it, we need to do a simulation run through the model, that follows the counterexample, by using the command '`spin -t model.pml`'. The printouts on the console include those from our designated print commands in the model, and thus, they constitute the wanted mix. All SPIN options, PAN compile time parameters and PAN execution time directives are listed in [12, chapters 18-19].

5.5 Tweaks

This paragraph discusses how we optimized the model, the specification and SPIN for our purposes.

Let us first consider the model. The tips on how to lower verification complexity from [12, chapter 11] are used as a guideline for improving the model:

First, the model was minimized by removing all non vital parts. The mixer has been excluded as it is redundant. Furthermore, computations are reduced to a minimum. The abstraction level is also chosen as high as possible, as we consider such big time intervals as one time step per section. Moreover, we went without making use of more than one process.

Second, the value ranges for each variable are chosen as small as possible. The maximum value range of integers is not used at all. In addition to that, we could give the execution a big boost by resetting temporal variables (that were only used for a local calculation) to zero after their usage.

Third, every deterministic sequence of statements was grouped into `d_step` and nondeterministic ones were grouped into `atomic` sequences, to further reduce the number of reachable states.

Finally, we could declare the tracks (which consume the biggest part of memory for each state) as `hidden`. That excludes them from the verification process which is desired here. When a track is loaded it is copied to the track field of the deck (which is visible for the verifier), so we can still access the tracks' data in the LTL formula; but only exactly when needed (i.e. when the track is playing). Thus, the number of tracks saved in a state is reduced from all to two; those loaded to the decks.

A specification can in average be verified faster if it is smaller. So it is lucrative to add properties that do not need a linear temporal description directly as constraints to the system. What is encoded as constraints is listed in table 5.1. Moreover, renouncing LTL's **X** operator improved the runtime enormously. In addition to that, simply factoring out the **G** operator from all small formulas led to a reduction of runtime by more than factor 50 in test runs. So having one big conjunction that is claimed to hold generally is much faster, than to have a big conjunction of formulas that each are claimed to hold generally.

SPIN offers several possibilities to reduce memory consumption during the model checking run. One of them is of particular interest to us. It is called **bitstate hashing** and can be activated with the PAN compile time directive `-DBITSTATE`. In bitstate hashing no whole states are stored, but only their hash codes. This voids completeness of the verification — the search for a counterexample is no longer exhaustive —

as hashcodes might be assigned repeatedly. However, this does not cause a problem for us: Every counterexample found still stays a valid counterexample. Besides, the non exhaustive search has never turned out to be the reason when no counterexample was found. Statistics on how much memory is saved by bitstate hashing can be found in paragraph 6.2. In addition to that, SPIN version 6.2.4 needed to be adapted to support bigger LTL formulas by increasing several field sizes.

It's supposed to be automatic, but
actually you have to push this button.

John Brunner

Chapter 6

Implementation

This chapter discusses the implementation of a user interface that enables us to easily perform a range of test runs. The concept of the application is to replace other music players on a daily basis and by that to yield superior aural pleasure. Due to a lack of time it is not implemented to work standalone, instead it remote controls a third party DJ software which will output the audio. In a standard digital DJing setup, the DJ would use a MIDI device to control the DJ software. Our application replaces the human component and the MIDI device in this setup. It calculates the mix and translates it to a sequence of MIDI messages that are sent to the DJ software.

Figure 6.1: The graphical user interface of the automatic DJ tool

6.1 The Graphical User Interface

The application is implemented in Java, the GUI in Swing. It has a modular design and consists of three separate components: The interface, the model creator and the remote control. We will touch on each components' implementation in this paragraph.

The **interface** provides the user with the ability to choose the tracks he wants to have mixed, to parameterize the mix synthesis and to finally initiate the play of the mix. It's main frame display can be seen in figure 6.1.

Automatic segmentation is not yet implemented. As a consequence, the sections have to be determined by hand, according to algorithm 1. Therefore, the GUI offers a fast and easy way to add new tracks and their segmentation to the library, by offering a mask in which all information that the model needs about the tracks can be entered. The inserted values are checked for plausibility and then saved. Each track entered in the mask is required to have at least an intro, a chorus and an outro section at the end. We differentiate between the name of a track (which can be set arbitrary and is just for representation within the GUI) and the filename of a track. The filename is only allowed to contain alphabetic characters and needs to have the extension `.mp3` or `.m4a`. The track library is saved as a XML file (short for Extensible Markup Language). Additionally, the library contains the paths to the audio files.

In the GUI, the library is represented as a list of tracks. Several tracks can be selected and thereafter two possibilities are offered to generate the mix: The 'play' button is used to synthesize a mix with default parameters. If none can be found, the parameters are loosened automatically and the search is repeated. This constitutes the fast and easy possibility to generate a mix, to be utilized by laymen.

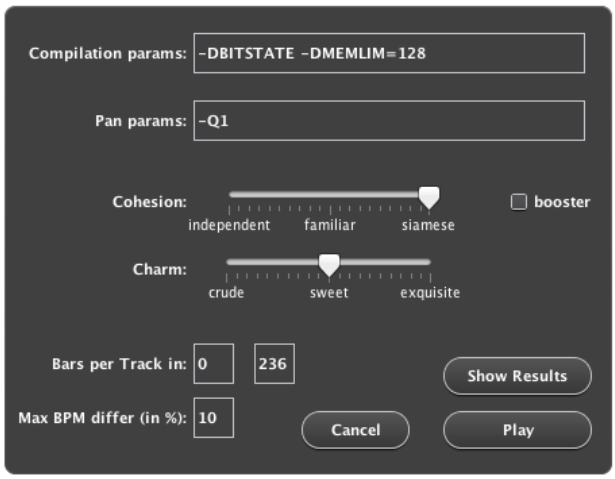


Figure 6.2: The advanced dialog allows for a detailed parameterization

The other possibility is to use the 'advanced' mode. A dialog (see figure 6.2) appears, that allows the user to adjust every parameter of the model and verification run. First, PAN's compilation and run time options are set. Then, the minimum percentage of overlapping segues ('cohesion') can be adjusted between zero ('independent') and 100 ('siamese'). The temporal logical formula 5.1 can be included (with the 'booster' checkbox), to force parallel playing during segues. Having this formula activated leads to a sensible reduction of runtime, compared with just setting the percentage of overlapping segues to 100%. Next, the minimum percentage of harmonic segues ('charm') can be set from zero ('crude') to 100 ('exquisite'). Moreover, the number of bars that may be played per track can be limited, as well as the maximum tempo difference between successive tracks. What is more, in the advanced mode we can also access the console output of the verification run.

The **model creator** interprets a specially prepared `model.pml` file. All parameterizable parts and those passages that depend on the specific tracks are excluded. First, the model creator generates a fully functional model file, depending on what was requested in the user interface and including the track specific parts. Then, it executes SPIN, compiles PAN and runs the verification. If a counterexample was found, the textual representation of the mix is extracted from the console printouts.

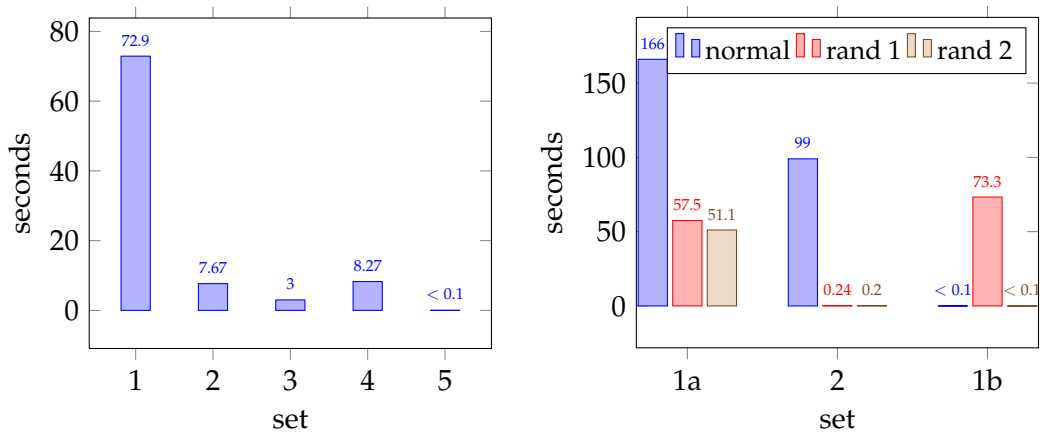
The **remote control** analyzes the textual mix and meanwhile creates a sequence of control actions that steer the DJ software. These actions are timed MIDI messages, which are commands that are sent through a MIDI bus; this virtual bus needs to be set up to connect our application with the DJ software. A so called MIDI sequencer plays this sequence of commands with precise timing. In advance, the DJ software is dependent on that it is told how to interpret those MIDI messages. Similar to creating a keyboard mapping we have to prepare a MIDI mapping. Additionally, the mixer states are deduced from the textual mix and also included into the control sequence. Currently the 'low' equalizer is linearly interpolated and the crossfader is simulated to follow the trajectory defined in paragraph 4.2. However, the tempo can not yet be manipulated, as the MIDI sequencer's execution tempo and the DJ software's playing speed are mandatory to precisely be the same. The remote control would need to be able to synchronize itself to the DJ software's tempo feedback, which it is not yet capable of.

In sum, limitations currently are: The track library cannot be extended as easy as choosing a music file and clicking 'add'. And we can only mix tracks with a similar tempo; however, this is not a serious issue, as most electronic dance music tracks have a similar tempo, anyway.

6.2 Analysis

As we consider a search space that is exponential in the number of tracks, does this approach actually deliver a mix in reasonable time? And if so, what is the maximum number of tracks that can be synthesized in a duration being applicable for everyday use? How does the runtime depend on the parameters? How much memory is needed? And finally, how good is the musical output?

For our test series we created a library of 50 tracks. Thereof seven tracks of the genre hip-hop, three classical house tracks, one pop music track, one dubstep track, and 38 electro house tracks. All major keys appear, but only two tracks have a minor key. 10% of the tracks differ by more than 10% from the tempo 128 BPM, but none of them is an electro house track. Among those, 87% differ by no more than 2% of 128 BPM. The average number of bars per track is 153, the minimum is 64 and the maximum duration is 264 bars. All test runs were performed on a 2.4 Gigahertz Intel Core 2 Duo processor.



(a) Comparison of different sets of 10 tracks each

(b) Randomized search

Figure 6.3: Runtime measurements

Chart 6.3a shows the dependency of the runtime, needed for finding a counterexample, on the set of tracks chosen. Each of the five sets measures ten tracks. The search was parameterized to find 100% overlapping and at least 50% harmonic transitions, the playtime per track was set to be between 60 and 180 bars, and the maximum tempo difference was fixed to 10%. The section step time model was used and parameters were held constant in each search. The parameter on maximum tempo difference will be set to 10% for all test runs in this paragraph. It does not affect the results at all, as only electro house tracks are used that all share approximately the same tempo. It is obvious from the chart, that the runtime dramatically depends on the track choice. But why is that so? First of all, it is easier to find a mix for some

sets of tracks than for others. For example when tracks musically fit better or more section sizes coincide. But the main reason for the enormous variability boils down to the requirement of 50% harmonic transitions. In the way we implemented the harmonic constraints SPIN does a trial and error search until it finds a mix that meets the harmonic requirement. This requirement can only be checked at the very end of the mix, so before that, otherwise desirable mixes are generated until one happens to meet the required minimum percentage of harmonic transitions. If a big number of mixes can be generated for a set of tracks, then many trials might be made until a valid mix finally is found. Thus, the big variations depend on how many mixes can be found in a set and also on a great deal of luck to quickly find a valid mix. Hence, it seems likely, that the runtimes would change fundamentally if the search was randomized.

In chart 6.3b we see the differences in the runtime with and without using random execution. We compare normal (top to bottom) resolution of nondeterministic options with randomized interpretations of nondeterminism and two different seeds (0 and 5). Set 1a and 2 of the sizes 10 and 12 tracks suggest that a randomized search shortens the runtime, possibly by factors up to 500. Parameters are assigned to generate 100% parallel and at least 50% harmonic transitions, as before, but playtime is now loosened to be within 50 and 220 bars. However set 1b shows, that randomized search not always shortens runtime, but might also extent it equally potent (here by more than factor 700). Set 1b consists of the same tracks as 1a but is not parameterized at all. To draw a conclusion, each search task probably has a random seed which minimizes its runtime, however, we cannot argue that overall one seed would be superior to the others, or that random search in general would be faster than the normal one.

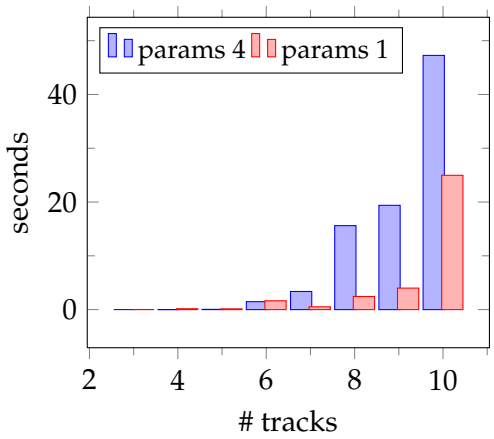
Due to the results presented in figure 6.3, we had to refrain from the idea that calculating average runtimes based on experiments (as conducted in figure 6.4 and 6.5) would deliver nice and definite results. In addition to the choice of tracks and luck as causes for the unpredictability also the chosen parameterization considerably affects the runtime.

The dependency of the runtime on the number of tracks is compared using four different parameterizations in figure 6.4. Three example sets of tracks were chosen and incrementally a larger number of tracks was picked out of them for each parameterization. The $\times$ symbol in the table means that using the current parameters and tracks no mix could be synthesized (i.e. the model checker could not find a counterexample). Typically for small numbers of tracks often no harmonic sequencing is possible. We arbitrarily defined 300 seconds to be the upper limit that a user of the automatic DJ tool would wait for the generation of a mix. Longer searches for counterexamples were aborted.

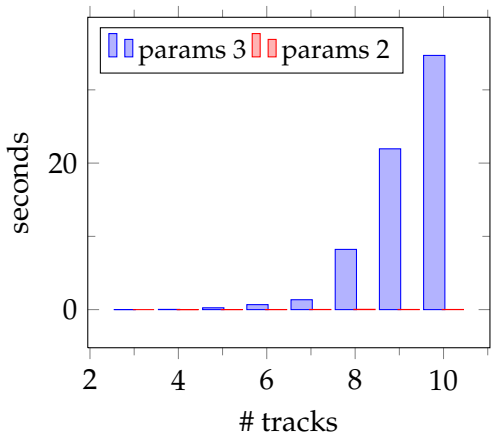
# tracks	params 1 set 1	params 1 set 2	params 1 set 3	params 2 set 1	params 2 set 2	params 2 set 3
3	0.03	×	0.01	<0.01	<0.01	<0.01
4	0.04	0.28	×	0.01	<0.01	<0.01
5	0.07	0.1	0.26	0.01	0.01	0.01
6	0.17	0.27	4.47	0.01	0.01	0.02
7	0.3	0.17	1.09	0.03	<0.01	0.02
8	4.12	0.58	2.63	0.09	0.01	0.02
9	7.27	2.07	2.66	0.03	0.01	0.02
10	48.1	2.2	24.6	0.03	<0.01	0.02
11	59.4	0.13	19.7	0.04	0.01	0.04
12	204	0.45	16.4	0.06	0.01	0.03
15	263	62	18.3	1.15	0.02	0.04
20	>300	>300	>300	0.12	0.03	0.02

# tracks	params 3 set 1	params 3 set 2	params 3 set 3	params 4 set 1	params 4 set 2	params 4 set 3
3	×	×	0.01	×	×	×
4	0.03	×	×	0.01	×	×
5	0.03	0.24	0.47	0.05	×	×
6	×	0.68	×	0.17	0.25	4.02
7	0.74	1.97	×	6.57	0.99	2.56
8	×	8.22	×	17.6	4.42	24.8
9	15.4	28.5	×	28.1	5.66	24.4
10	16.3	53.1	×	48.2	8.19	85.4
11	23.4	177		233	10.7	38
12	63	>300		>300	41.1	37.8
15	>300				177	97.8
20					>300	>300

(a) Detailed results for different parameterizations (in seconds)



(b) Comparison of parameterization 1 and 4



(c) Comparison of parameterization 2 and 3

Figure 6.4: Runtime measurements for different parameterizations

- **Params 1** requires 100% of the transitions to be overlapping (formula 5.1 is used to speed up the synthesis), 50% of the transitions to be harmonic and all tracks have to be played for between 50 and 220 bars. We define this parameterization to be the **default parameterization** that delivers mixes having a good tradeoff between aural quality and calculation time.
- **Params 2** again requires 100% of the transitions to be overlapping and that all tracks have to be played for between 50 and 220 bars, however it does not put any constraints at all on the harmonic compatibility of the segues.
- **Params 3** is stricter than the default parameterization in that a track has to be played for a duration between 80 and 160 bars.
- **Params 4** differs from the default parameterization in that it requires the higher percentage of 60% of all segues to be harmonic.

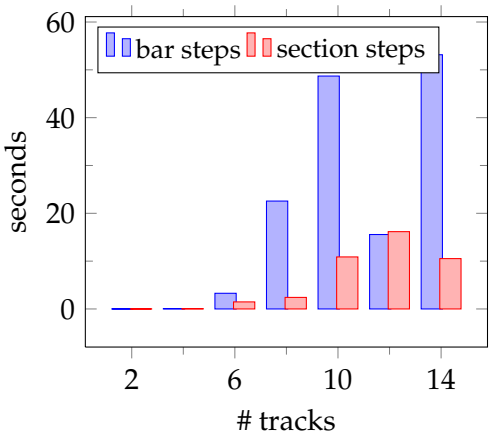
The dependency of the runtime on the number of tracks is often not harmonic, which is caused by an unsteady difficulty of the search, depending on the tracks and on the number of trials needed to find a mix satisfying the harmonic requirements. Charts 6.4b and 6.4c show the mean values of the runtimes over the three sets for each number of tracks. We can see the influence of the harmonic requirement in chart 6.4b. The increase in harmonic segues by 10% lead to an approximately doubled runtime. If we compare 6.4b and 6.4c we learn that the imposed restriction of the duration a track is played does not increase the runtime as much as the 10% increase in harmonic segues. Furthermore, from chart 6.4c we learn that, depending on the parameters chosen, the dependency on the number of tracks is not always exponential (at least for the examined problem sizes). Moreover we make the important finding, that for track sizes below 10 tracks the synthesis can be expected to need an agreeable time of below one minute. Thus, this approach has proven to be applicable for everyday use despite the underlying exponential search problem. For parameterization 2 all but one value are below 0.1 seconds, so the model's constraints clearly did a good job in minimizing complexity (nearly to a constant complexity for the examined problem sizes). However, the maximum number of tracks that can be synthesized in bearable time seems to be limited by less than 20 tracks for typical parameterizations.

In 20 out of 135 test runs no mix was found, which was caused by a too restrictive set of parameters for the tracks to be mixed. General reasons for non-monotonic trends that occur in the tables are a lucky or unlucky trial and error search, or a constellation of keys that actually made the search task easier (for keys 1A and 4A no harmonic segue can be found, however, if a track with key 2A is added it becomes possible), as it also was discussed earlier in this paragraph.

The measurements for the comparison of runtime between the time step implementations (bar or section sized steps) are presented in figure 6.5 (for all other test runs in this paragraph entirely the section step model was used). Set 1 was unparameterized except for the requirement that the mix has to be 100% harmonic. Set 2, 3 and 4 were parameterized according to the default parameterization. As expected after the results of figure 6.3, the function of the number of tracks is not always monotonic. Still we can point out the key characteristic of the measurements: In average, bar sized time steps needed significantly (factor 3.45) more runtime than section sized time steps in these measurements. Furthermore, a mix was found by section sized time steps in four cases, were none could be found by the bar sized time steps model. Summarized, these measurements justify the decision of using section sized time steps as default.

	bar steps				section steps			
# tracks	set 1	set 2	set 3	set 4	set 1	set 2	set 3	set 4
2	<0.01	×	0.02	0.01	0.01	×	<0.01	0.01
4	0.04	×	×	0.11	0.13	×	×	<0.01
6	0.98	11	0.06	1.07	0.03	0.75	4.28	0.84
8	3.47	48.8	4.58	33.4	0.12	5.16	2.67	1.71
10	56	71.6	18.5	×	0.16	9.9	25.1	8.36
12	29	×	4.13	×	0.33	40.4	19.6	4.33
14	106	×	0.32	×	1.38	13.1	17.1	×

(a) Detailed results for both implementations using four test sets (in seconds)



(b) The mean values of the results above

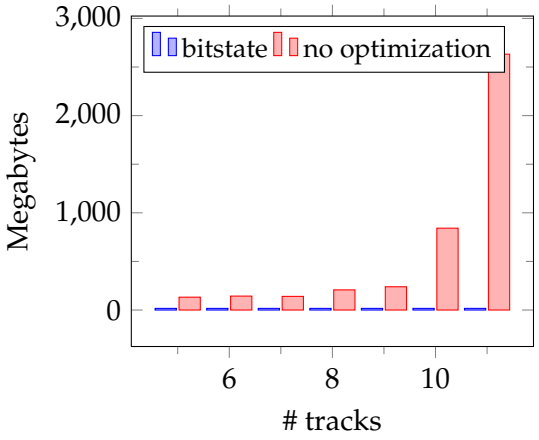
Figure 6.5: Runtime measurements comparing the two time step implementations

Figure 6.6 shows the results of the measurements that compare the memory consumption with and without using bitstate hashing. The search was parameterized according to our default set of parameters from the charts above. Without the use of memory optimization techniques we quickly (in fact exponentially) run to the

physical boundaries of a personal computer. However, using bitstate hashing the memory consumption never exceeded 17 Megabytes; 16 Megabytes thereof occupied by the hash table. As further the runtime is not noticeably influenced by this technique and as it did not result in unsuccessful searches, we use this optimization as default and obtain an almost constant memory consumption. In particular, it was used for all of the test runs above. The memory consumption might even be further minimized by downsizing the hash table. Chart 6.6b shows the mean values for the three sets being listed in table 6.6a.

	bitstate hashing			no optimization		
# tracks	set 1	set 2	set 3	set 1	set 2	set 3
5	16.93	16.93	16.93	130.88	136.74	129.12
6	16.93	16.93	16.93	134.59	163.40	132.15
7	16.93	16.93	16.93	138.69	153.53	128.93
8	16.93	16.93	16.93	251.78	218.96	150.90
9	16.93	16.93	16.93	334.00	254.12	133.03
10	16.93	16.93	16.93	1526.97	818.48	180.78
11	16.93	16.93	16.93	3358.91	2451.39	2085.76

(a) Memory usage with and without optimization techniques (in Megabytes)



(b) The mean memory usages for the results above

Figure 6.6: Memory measurements

Day-to-day testing showed that mixes of 10 tracks or less with default parameters can usually be found in less than one minute or even much quicker. If the harmonic value and playtime restrictions are set reasonably, under consideration of the tracks given, a mix is found in 10 out of 10 times. If those parameters are tightened, quickly it no longer becomes possible to synthesize a mix. With no parameter restrictions at all the complete library of 50 tracks was mixed in 51 seconds.

Before the model checking can be conducted, SPIN needs to analyze the model file

and to create the PAN program. For two tracks this takes 0.21 seconds (the mean value of three measurements with different tracks), 0.24 seconds for 10 tracks and 0.75 seconds for 50 tracks. So the time SPIN needs is negligible.

The aural experience is as expected: The segues are done between reasonable sections of the tracks. They are performed technically correct and smooth, and thus, they sound proper. Overlapping segues are preferred over cuts, so we normally demand 100% of the segues to be played in parallel. This is not because cuts would be defective, but they are simply inferior to overlapping segues in lending a mix its DJ charm.

Table 6.1 lists the results of an extensive evaluation of three automatically synthesized mixes. Each set of tracks that was mixed consisted of 16 tracks and thus the resulting mixes had 15 segues. The sets of tracks were chosen completely arbitrarily and were parameterized according to the default parameterization. Every segue was rated with +, ∘ or - where + means that it was good/as desired, ∘ means that it had minor flaws that did not affect listening quality a lot and - means that it showed flaws that caused discomfort. The type of each segue is stated according to the types introduced in paragraph 4.2. They are well distributed: 17 times type 1, 17 times type 2 and 11 times type 4 (type 3 and 5 are not possible for 100% parallel segues).

In 42 out of 45 times, the resulting segues were enjoyable to listen to, yet of course they did not arouse attention or boost energy as an experienced human DJ's performance would. Out of those 42 segues, 34 were optimal and seamless and 8 showed minor flaws not causing discomfort:

- In 2 cases the actual energy curve did not coincide with the energy curve expected by the listener. In the first case this was caused by the, relative to the new track, high energy of the outro of the old track. In the second case this was caused by our level of abstraction that omits the differentiation between interludes and build-ups. The build-up was followed by a low energy section.
- In 4 cases the melodic components of the combined tracks did not harmonize. However, in 3 of those 4 cases the keys of the tracks should have been compatible according to the results of the computerized analysis. It seems likely that the analysis results were faulty, as a second review with another analysis tool produced different and incompatible keys.
- The last 2 flaws were caused by the automatic beat detection of the DJ software. The phase of one track was displaced by half a beat. As each track (except the last and first of a mix) is involved in two segues, this affected two transitions.

Transition		Rating +/○/-	Faults	Type	Duration (in bars)
set 1	1	+		1	8
	2	+		1	16
	3	+		1	32
	4	○	unexpected breakdown of energy	2	32
	5	+		4	8
	6	+		2	8
	7	○	clashing despite compatible keys	4	16
	8	+		1	8
	9	○	clashing despite compatible keys	4	16
	10	+		4	8
	11	+		4	8
	12	○	clashing despite compatible keys	4	8
	13	+		2	8
	14	+		2	16
	15	+		2	16
set 2	1	+		4	8
	2	+		2	32
	3	○	breakdown of energy after a build-up	4	16
	4	○	faulty automatic beat detection of track:	2	8
	5	○	phase wrong by 1/2 beat	1	8
	6	-	automatic beat detection of track failed	1	32
	7	-	(despite that transition would be good)	2	16
	8	+		1	8
	9	+		4	16
	10	+		2	8
	11	+		1	8
	12	+		2	16
	13	+		1	8
	14	+		2	16
	15	+		2	8
set 3	1	+		2	16
	2	+		1	16
	3	+		1	32
	4	+		1	16
	5	+		1	8
	6	+		1	16
	7	+		2	16
	8	+		1	8
	9	+		1	8
	10	+		2	16
	11	-	clashing melodics, incompatible keys	2	16
	12	+		1	8
	13	+		4	8
	14	○	clashing melodics, incompatible keys	2	16
	15	+		4	8

Table 6.1: Aural evaluation of three synthesized mixes of 16 tracks each

In 3 cases the quality was affected by severe flaws:

- In 2 cases again the cause was that the automatic beat detection of one track failed. This time not even the tempos were aligned correctly, which led to considerable listener discomfort.
- In the other case, two tracks with incompatible keys were combined in parts with a rich melodic content. The obvious reason for that is, that we demanded only 50% of all transitions to be harmonic.

The faults in the beat detection were bad luck as they do not normally happen. If we exclude those cases from the evaluation and furthermore those, in which the key was analyzed wrong, we obtain 34 out of 38 good transitions (= 89.5%). And only 1 out of those 38 segues caused significant discomfort (= 2.6%). A slight improvement could be achieved by changing the crossfader movement, as it sometimes appears to be too harsh, and by extending the equalizing. The mean duration that two tracks were played simultaneously during a segue is 13.87 bars which equals impressive 26 seconds at a tempo of 128 BPM (= 32 bars per minute). Consequently, the results assured the capabilities of the proposed mixing technique and realization. What is more, the automatic DJ tool proved to be robust with parameters not optimized for aural quality, as the parameters constituted a well-balanced tradeoff between aural quality and synthesis complexity and still the evaluation was affirmative. The three mixes were synthesized in a mean runtime of 98 seconds while using 17.12 Megabytes of memory at the most.

> Science is what we understand well enough to explain to a computer; art is everything else.
>
> *Donald E. Knuth*

Chapter 7

Conclusion and Future Work

7.1 Conclusion

We live in an era of rapid technological change. More and more mechanical jobs are continuously replaced by robots. But also the automation of tasks via computer software continues to advance. From 1998 on, attempts were made to automate DJing. These strive for replacing human DJs while preserving the listening experience. Bars gain from that as they no longer have to pay a DJ and save a lot of money, people can listen to mixed music on the go and radio stations can play music like a DJ for 24 hours every day.

This thesis made a step forwards the automation of the task of DJing, for both, its technical and musical components: We developed a conclusive system that aims at accurately imitating a DJ's performance. Other approaches to this task have been presented and shown to be intermittent, when it comes to respecting the musical structure of electronic dance music; also they constitute chiefly algorithmic solutions. We proposed a new human-like approach. Using modern model checking methods, borrowed from theoretical informatics, we were able to abstract away from a pure algorithmic approach and could solve a real life problem: How to manipulate a DJ system to obtain a mix while acting in a similar way to how a human DJ would. Therefore, we formally modeled the abilities of a DJ system and formalized a set of rules which allowed only specific actions. This separated description of the task allows future users to exchange the formula with another one, that defines a new mixing technique. First, the model was specified mathematically, to make it easily portable. Subsequently, it was implemented in a concrete modeling language. Using model checking enables us to synthesize a mix using one all-encompassing method. This guarantees a more optimal result than splitting the task into first finding a sequencing and then the segues, as other approaches do. But it not only beats the

other patchwork-like solutions, as model checking constitutes a popular research field, we also profit from the existence of fast and efficient search algorithms which are still being developed further. At the same time, an unusual application field for model checking was developed, as the disciplines of DJing and verification were brought together.

The model and specification were realized after a prior study of the structure of electronic dance music and of DJing. We looked at basic concepts in DJing, at the hardware used and at specific techniques. Afterwards, we formulated a set of default rules on what DJs do and implemented it. The other presented approaches could not be built on, as they abstracted music in different ways or their methods disagreed with our intents. Moreover, often only little technical details were published.

As it is common for search problems, the worst case runtime depends exponentially on the problem size. However, we showed that the idea of using model checking and the practical implementation actually are applicable. For unexceptional numbers of tracks and reasonable parameter restrictions the runtime proved to be satisfying and sometimes even fast. Unfortunately it turned out to be quite unpredictable, which surely would not sell well in a commercial application.

A graphical user interface, aiming at making generating and playing a mix as simple and intuitive as possible, was designed. But just after it was set up to work with a third party DJ software, the application is easy to use. Remote controlling DJ software has the positive aspects that we did not have to re-implement its capabilities and that we did not have to care about synchronizing the tracks. Furthermore, DJs profit from this approach, as they can easily let the application take over control. It is designed in a way that the three parts user interface, model creator and remote control can separately be replaced, if required. From a musical point of view, the audio output was evaluated predominantly positive, under the judgment criterion that segues appear seamless and smooth. This impression is further supported by the long duration of the segues, which leaves its competitors behind.

7.2 Future Work

The first thing that comes to ones mind when using the automatic DJ application is, that adding tracks to the library is painstaking. We already laid the foundation on how to implement an appropriate automatic segmentation algorithm, but it has not been realized yet.

The second weakness is, that the application is not standalone. Two alternatives are nearby: Either we strive for developing an easy-to-use standalone application (at the best a mobile application that outsources the synthesis task to a server and is able to

work with the user's music library), or we develop a plugin for a common audio player or DJ software (by that users could continue to use their favorite players and still enjoy DJ like playback).

Many different model checking methods exist that could improve the capabilities of the synthesis. For example the utilization of symbolic model checking would enable us to search much larger sets of states, and we could address the shortcomings of the model: The description of sections should be added information about their melodic content and exact structure, so that we can enhance the mixer controls and section matching by using finer rules. Likewise, bridges should be broken down into interludes and build-ups depending on their energy trends, to rule out the ambiguity that currently prevails. Having a more capable search at its disposal would also render it possible to choose the tracks that it mixes on its own out of a library, on the basis of one or several query tracks. By that, it would not be forced to combine incompatible tracks and could synthesize *ideal* mixes.

Future implementations could make use of similarity matrices in a reverse way: Exceptionally bad fitting transitions could be forbidden. Similarly, whole tracks that do starkly differ from the rest of the mix in a sonic quality such as timbre could be excluded. This backwards usage would merely disallow a number of mixes but not affect the synthesis otherwise.

Last but not least, the tool could be upgraded to an all-in-one entertainment system: Additionally to the music it could also control the effect and light system in a bar or club. The model would need to be extended by the capabilities of the light system, possibly by using a second process, and a new set of rules would have to be developed. As a result, the counterexample path would constitute a sequence of pairs of the light system's and DJ system's state.

Acknowledgements

I would like to express my deepest gratitude to Bernd Finkbeiner for providing me the possibility to write this thesis, for supporting me in formulating the task and building a bridge between music and model checking, and for always being willing to listen to my problems. Furthermore, I would like to thank Meinard Müller for helping me to find my way in the jungle of audio segmentation. Also, I would like to thank my loved ones for keeping me harmonious throughout the entire time, for their stimulating suggestions and for helping me reading and correcting my thesis. I would like to thank Pascal Berrang and my other fellow students for their feedback, which I highly esteemed. Special thanks go to Mischa Jungmann who helped me proofreading and alone was able to spot a countless number of mistakes.

Bibliography

[1] M. A. Alonso, G. Richard, and B. David, "Tempo and beat estimation of musical signals.," in *ISMIR*, pp. 158–163, 2004.

[2] T. H. Andersen, "Mixxx: towards novel DJ interfaces," in *Proceedings of the 2003 conference on New interfaces for musical expression*, NIME '03, (Singapore, Singapore), pp. 30–35, National University of Singapore, 2003.

[3] C. Baier and J.-P. Katoen, *Principles of Model Checking*. The MIT Press, 2008.

[4] W. Chai and B. Vercoe, "Music thumbnailing via structural analysis," in *Proceedings of the eleventh ACM international conference on Multimedia*, MULTIMEDIA '03, (New York, NY, USA), pp. 223–226, ACM, 2003.

[5] E. Clarke, O. Grumberg, and D. Peled, *Model Checking*, p. 14. University Press Group Limited, 1999.

[6] D. Cliff, "Hang the DJ: Automatic sequencing and seamless mixing of dance-music tracks," tech. rep., HP Laboratories Bristol, 2000.

[7] D. Cliff, "hpDJ: An automated DJ with floorshow feedback," tech. rep., HP Laboratories Bristol, 2005.

[8] M. Cooper and J. Foote, "Automatic music summarization via similarity analysis," in *Proc. of Int. Symposium on Music Information Retrieval*, pp. 81 – 85, 2002.

[9] R. Curticapean, "Clustering-based audio segmentation with applications to music structure analysis," Bachelor thesis, Saarland University, 2009.

[10] EVAR Advisory Services, "Dance-onomics, the economic significance of edm for the netherlands," October 2012.

[11] M. Goto, "Smartmusickiosk: music listening station with chorus-search function," in *Proceedings of the 16th annual ACM symposium on User interface software and technology*, UIST '03, (New York, NY, USA), pp. 31–40, ACM, 2003.

[12] G. J. Holzmann, *Spin model checker, the: primer and reference manual*. Addison-Wesley Professional, first ed., 2003.

[13] G. J. Holzmann, "The model checker spin," *IEEE Trans. Softw. Eng.*, vol. 23, pp. 279–295, May 1997.

[14] H. Ishizaki, K. Hoashi, and Y. Takishima, "Full-automatic DJ mixing system with optimal tempo adjustment based on measurement function of user discomfort," in *Proceedings of the 10th International Society for Music Information Retrieval Conference, ISMIR 2009, Kobe International Conference Center, Kobe, Japan, October 26-30, 2009* (K. Hirata, G. Tzanetakis, and K. Yoshii, eds.), pp. 135–140, International Society for Music Information Retrieval, 2009.

[15] T. Jehan, *Creating Music by Listening*. PhD thesis, School of Architecture and Planning, Massachusetts Institute of Technology, September 2005.

[16] H.-Y. Lin, Y.-T. Lin, M.-C. Tien, and J.-L. Wu, "Music paste: Concatenating music clips based on chroma and rhythm features.," in *ISMIR* (K. Hirata, G. Tzanetakis, and K. Yoshii, eds.), pp. 213–218, International Society for Music Information Retrieval, 2009.

[17] A. Marsden, "Timing in music and modal temporal logic," tech. rep., Lancaster Institute for the Contemporary Arts, Lancaster University, 2007.

[18] B. Pipiorke-Arndt, *Digital DJ-ing: Tipps, Tricks & Skillz für Discjockeys*. Studiopockets - Fachwissen kompakt : Audio, Quickstart Verlag, 2009.

[19] E. Scheirer, "Tempo and beat analysis of acoustic musical signals," *J. Acoust. Soc. Am*, vol. 103, pp. 588 – 601, Jan. 1998.

[20] Y. Vorobyev, E. Coomes, and B. Murphy, *Beyond Beatmatching: Take Your DJ Career to the Next Level*. Mixed In Key, 2012.

[21] S. Wenger and M. Magnor, "Constrained example-based audio synthesis," in *Proceedings of the 2011 IEEE International Conference on Multimedia and Expo*, ICME '11, (Washington, DC, USA), pp. 1–6, IEEE Computer Society, 2011.

[22] Z. Zuo, "Towards automated segmentation of repetitive music recordings," Master's thesis, Saarland University, 2011.

Online-Sources

[23] Bundesverband-Musikindustrie, "Musiknutzungs statistiken 2012." `http://www.musikindustrie.de/jahrbuch-musiknutzung-2012/`. last checked: July 25, 2013.

[24] C. Cartledge, "Key detection software showdown: 2012 edition." `http://www.djtechtools.com/2012/01/26/key-detection-software-showdown-2012-edition/`. last checked: July 25, 2013.

[25] Columbia University, "Meapsoft, a program for doing automatic segmentation." http://www.meapsoft.org. last checked: July 25, 2013.

[26] DJ Mandrick, "Understanding dance music structure." http://www.djmandrick.com/html/djtutorials/understanding-dance-music.htm. last checked: July 25, 2013.

[27] Fidelity Media, "Megaseg." http://www.megaseg.com/about.html. last checked: July 25, 2013.

[28] Z. O. Greenburg, "The world's highest-paid DJs 2012." http://www.forbes.com/sites/zackomalleygreenburg/2012/08/02/the-worlds-highest-paid-djs/. last checked: July 25, 2013.

[29] T. Haste Andersen and K. Haste Andersen, "Mixxx, an open-source DJ software." http://www.mixxx.org. last checked: July 25, 2013.

[30] G. J. Holzmann, "Spin webpage." http://spinroot.com/. last checked: July 25, 2013.

[31] C. L'Hopital, "Automatically performed crossover between two consecutively played back sets of audio data." http://patent.ipexl.com/EP/EP0932157.html. last checked: July 25, 2013.

[32] P. Lamere, "The infinite jukebox." http://musicmachinery.com/2012/11/12/the-infinite-jukebox/. last checked: July 25, 2013.

[33] Native Instruments, "Traktor DJ software." http://www.native-instruments.com/en/products/traktor/dj-software/traktor-pro-2/. last checked: July 25, 2013.

[34] Ots Labs, "Otsav's intellifade mixing engine." http://www.otslabs.com/intellifade/. last checked: July 25, 2013.

[35] O. Parviainen, "Soundtouch audio processing library." http://www.surina.net/soundtouch/. last checked: July 25, 2013.

[36] Pioneer, "CDJ player." http://pioneerdj.com/english/products/player/. last checked: July 25, 2013.

[37] Pioneer, "Mixtrax." http://www.mixtrax-global.com/en/lp/mixtraxapp.html. last checked: July 25, 2013.

[38] PRWeb, "Barjock - the software that manages background announcements and music for your bar, pub or restaurant." http://www.prweb.com/releases/background_music_software/announcements/prweb10189410.htm. last checked: July 25, 2013.

[39] Serato, "Scratch live DJ software." http://serato.com/scratchlive. last checked: July 25, 2013.

[40] The Echo Nest, "Solutions." http://echonest.com/solutions/. last checked: July 25, 2013.

[41] Y. Vorobyev *et al.*, "Mixed in key." http://www.mixedinkey.com. last checked: July 25, 2013.

[42] Zytrax, "Equalization, metering and the FFT." http://www.zytrax.com/tech/audio/equalization.html. last checked: July 25, 2013.