Automatic Trance Music Generation and Automatic Melodic Variation
as Aesthetic Music?

Marvin Beese, mabeese@uni-potsdam.de, 786300

University of Potsdam

Abstract. In Answer Set Programming music is a field of application with the ability of the generation of music using constraints and analyzing the result. With "Armin" it is possible to generate Trance music and with "AIM" melodic variation can be applied to simple melodic lines. In the analysis of these two systems, the overall consideration of the aesthetics of music is also important and will be thematized.

Keywords: Answer Set Programming · Anton · Armin · AIM · aesthetics.

1 Introduction

A field of Artificial Intelligence (AI) is automatic music composition with Answer Set Programming (ASP). As this is not the only way of creating music programmatically, the aesthetic quality of rule-based and generic systems can be compared. With the Anton-System [1] as a foundation, the automatic generation of Trance music is possible with the Armin-System using ASP. For musical variation using a melodic variation engine, the System Alterations in Music (AIM) was developed with ASP. This summary is based on the articles "An aesthetic comparison of rule-based and genetic algorithms for generating melodies" by Andrew R. Brown [2], "Armin: Automatic Trance Music Composition using Answer Set Programming" by Flavio Omar Everardo Pérez and Fernando Antonio Aguilera Ramírez [3] and "A Logical Approach for Melodic Variations" by Flavio Omar Everardo Pérez [4]. First, I introduce the aesthetics of generated music using rule-based and generic systems, then I give a firm introduction into Automatic Trance Music Composition. Finally, I look into a Logical Approach for Melodic Variations using ASP.

2 Aesthetics of generated music using rule-based and generic systems

2.1 Musical Aesthetics

For the generation of music in Artificial Intelligence (AI), one of the main questions for the aesthetics of algorithmic generation of music is what musical tendencies are used and how they complement each other when combined. When

trying to answer this question it seems that music can play an important role in the area of AI, as the musical understanding is generally speaking a mode for aesthetic understanding. This can be valid and useful for the measurement of algorithmic success, especially if it comes to aesthetic judgements as a reliable guide to achieve the ultimate goal of software to try to break free conceptually. Besides rule-based systems, where rules formulate constraints for building and limiting melodic material, generic algorithm processes (GA) can be regarded as evolutionary systems. Here variation happens through mutation and the selection of generated elements is dependent on a fitness function.
Aesthetic judgement follows the heuristics of finding glimpses of elegance and novelty in melodies, while trying to minimize incoherence and boredom, still affection or emotional expressions are not regarded, as this is more a philosophic topic of finding aesthetic values. Particularly, tonal and rhythmic coherence, stability of melodic contour and balance of repetition and variety are the main aspects of examination.

2.2 Rule-based processes and genetic algorithm processes

Rule-based processes use explicit encoded knowledge, which are rules derived from texts to assist beginner composers regarding pitch selection, rhythmic construction, repetition and melodic contour. Using rules, a predictable behavior is expected, which doesn't have to be deterministic.
Genetic algorithm processes on the other hand are inspired by evolutionary biology, where outcomes are produced with more or less a blind chance. The production of new elements depends on combining elements from successful outcomes. If there are changes for improvement, mutation comes into existence for variations of the melody, where an individual melody tries to improve its fitness score. Mutations can both be of an evolutionary or musically specialized kind. In evolutionary mutations, the pitch of one random note will be changed for every generation or with split and merge, the melody will be divided at beat boundaries and recombined with sections from other melodies. However, musically mutations follow more compositional rules for variations, which are not regarded in this summary.
In a study of Andrew R. Brown [2], initial populations of melodies with randomized pitch and rhythmic value as well as melodies generated by rules were used for comparing the unprocessed music with different stages of mutation regarding the aesthetic judgement.

Table. An overview of the results of assessment by aesthetic judgement.

Beginning state	Unprocessed	Fittest unmodified	Evolutionary mutations	Musical mutations	Combined mutations
Random	★	★	★	★★	★
Rule-based	★★★	★★★	★★	★★★★	★★★★

The results of unprocessed and mutated music with a rule-based initial population were nearly comparable and generally speaking well-formed melodies, even

though novelty is rare. Furthermore, evolutionary mutations tend to reduce coherence of form with minimal gain in elegance or novelty. This undermines the reliable capabilities of rule-based systems, and will be further examined with the automatic generation and logical variations of music using Answer Set Programming.

3 Armin: Automatic Trance Music Composition using Answer Set Programming

3.1 Background of computer-aided Composition and Trance Music

When music pieces get created from synthetic instruments with lifelike sounds or sound-effects, there is no recording of each instrument in studios, instead the song gets rendered. In computer-aided composition (CAC), which is a form of algorithmic composition, the ideas for creating or editing music will be mapped to suitable software before the rendering of the final song happens. For the composition of music, musicians as well as scientist try to answer the question, where the next note comes from and beyond, what the duration of this note is respective the music genre. Using CAC, when trying to imitate the production of a musical piece, the final piece must be similar to other known pieces. For that, questions like what rules should the music follow, how similar is it to other music and how many and which notes should vary from known pieces must be applied in the production of the software.
Trance music is a genre of electronic dance music. Therefore, characteristic is a percussion frequency of 130-140 beats per minute (bpm) with a time signature of 4/4, which means that in a bar/measure there are 4 (quarter) measures. Here each beat has a kick and on the second and fourth beat is a snare drum or a clap sound. The pace of the music changes every second, fourth or eighth bars and with that pace change there comes also a sound change of the drum or a minor instrument change, which is referred as a progression. It is important, that a rhythmical change is not a random event but is more strategically placed, mostly leading to an increasing intensity of the rhythm. Every now and then there are no beats in the song, which leads to more importance of synthetic sounds, longer chords and generally slower paces, these are breaks or breakdowns. Very important in Trance music are loops, which are repetitions in the song with some variations to avoid musical monotony and to include more pleasant rhythm, for that instruments as brass drum or snare claps are often used.

3.2 Armin System

Armin is a Trance music composition system, named after the trance music producer Armin van Buuren, which uses Answer Set Programming (ASP) to benefit from the modelling simplicity and the high-performance solving. Here, the modelling of one or more instruments is possible, where each instrument contains its own set of rules. The System is based on Anton 2.0.0, a melodic, harmonic and

rhythmic composition system [1] with up to 4 voices in the renaissance composition system. Anton can also be used for diagnosing errors given an input file and for completing a piece. For latter, Anton runs the phases of building the program, running the solver and interpreting the results. With Anton as a basis, Armin is able to do musical sections chaining, where the introduction leads to a verse, the verse to a chorus and the chorus to a breakdown.

Architecture and Components In Anton, the two main program parts are the score generator with the driver and assembly file and the sound generator for the subsequent generation of audio. The driver is regarded as the main file, which includes some input parameters like the bpm, fundamental for the musical scale, possible mode for the melodic composition and the number of parts or sections of the piece. In the assembly file, the order and the appearance frequency of parts in the song is declared, with the dependencies between parts and the possible behavior over time.
In *arminAssembler.lp* the model is defined over timestep T, starting with 1, where the intro part has to be played: `playState(intro,1):-part(intro)`. In the following timesteps $T+1$, either the verse, chorus or the breakdown will be selected if the statesNumber (SN) indicates, that the song will not end with that:

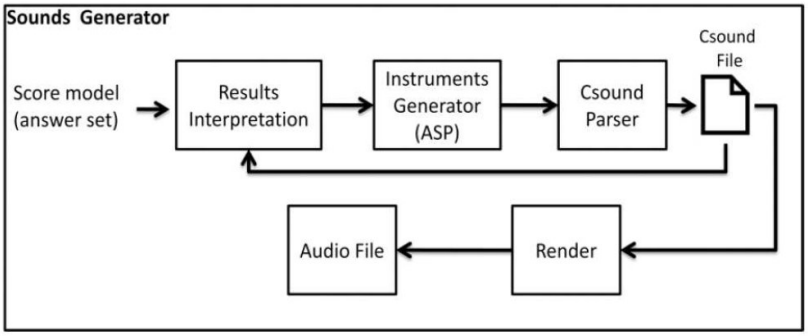
```
1{playState(verse, T+1), playState(chorus,T+1),
    playState(breakDown,T+1)}1:- playState(P,T),
    timeScore(T), statesNumber(SN),T<SN-2.
```

For the end part, the outro will be selected as the final state when *statesNumber* is reached: `playState(outro,SN):-part(outro), statesNumber(SN)`. Some additional constraints prevent consecutive verses, three consecutive played parts, allow for special modes before the outro and declares a soft chorus, as stated in the following fragment of *arminAssembler.lp*.

```
%%The verse(s) cannot be played in the time T and in the time T+1
:- playState(verse, T), playState(verse,T+1).

.%%No part can be played 3 times in a row
:- playState(P,T), playState(P,T+1), playState(P,T+2).

%%Before outro can be played chorus or breakdown
1 { playState(chorus,T-1), playState(breakDown,T-1) } 1 :- playState(outro, T).

%%The second breakDown in a row equals a soft chorus
playState(softChorus,T+1) :- playState(breakDown, T), playState(breakDown,T+1).
```

For the generation of audio, the sound generator interprets the results by sending each instrument to the instrument generator, where the model of the instrument gets created. The file from the instruments generator is the input file for the Csound parser, which maps the results to the Csound format, which is rendered as a WAV audio-file. The program flow-diagram of the sounds generator can be

viewed below.



Melodic and Percussion Instruments The Armin system uses two classes of instruments, the melodic instruments and the percussion instruments. The melodic instruments usually play during the chorus and the breakdown section as long behavior instruments like strings. As the time signature in Trance music is 4/4 which can be filled with either a whole note (32 pulses) or two half notes (each 16 pulses), a measure is defined as 32 pulses with `pulseMeasureLimit(32).` and the duration of a half note or a whole note with `longDurations(16;32).` in *progression.lp*. In our accompanying example (*Figure 5. Example of an 8 bar configuration*), melodic instruments perform 8 measures, which is defined with `lastMeasure(8).` The following code fragment declares that either a half or a whole note can follow the current bar and that half notes must come in a pair.

```
%% If the Duration in the current state is 32, the next state can be 32 or 16
1 { durationMeasure(0,D1,M+1,C+1)
                : longDurations(D1) } 1 :- durationMeasure(0,DR,M,C),
                                DR == 32, lastMeasure(LM), M + 1 <= LM.

%% If the Duration in the current state is 16, the next state must be 16
1 { durationMeasure(16,D1,M,C+1)
          : longDurations(D1) } 1 :- durationMeasure(0,DR,M,C), DR == 16.

%% If the Duration in the current and in the previous state the durations were 16,
%% the next state can be 32 or 16
1 { durationMeasure(0,DR,M+1,C+1) : longDurations(DR) } 1 :- durationMeasure(16,16,M,C),
                        durationMeasure(0,16,M,C-1), lastMeasure(LM), M + 1 <= LM.
```

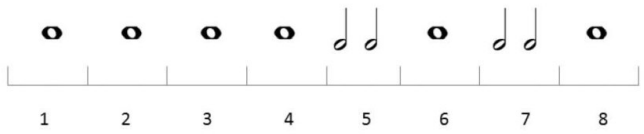


Figure 5. Example of an 8 bar configuration

In a Trance music progression, a change or repetition of a note depends on the

count of the measure. This means, that if there is an odd measure, there must be a change of the note and if there is an even measure, both a change or a repetition of the note can happen. If a measure is divided into two halves, then the second note must have a different note than the first one to have a better difference to a whole note. Furthermore, if the last measure consisted of halves, then the next measure must have a note change. These constraints are defined in the following code fragment of *progression.lp*.

```
%% It changes by stepping (moving one note in the scale)
%% or leaping (moving more than one note)
%% These can either be upwards or downwards

%% If the measure number is odd... Change
1 { changes(P,T,C) } 1 :- partTime(P,S,T,C), partTimeMax(P,TM), T != TM, (T mod 2) != 0.

%% If the measure number is even... change or repeat
1 { changes(P,T,C), repeated(P,T,C) } 1 :- partTime(P,S,T,C),
                                           partTimeMax(P,TM), T != TM, (T mod 2) == 0.

%% If the measure is divided into two 16 notes... change the second one
1 { changes(P,T,C) } 1 :- partTime(P,S,T,C), partTimeMax(P,TM), T != TM, S == 16.

%% If the previous measure is divided into two 16 notes... change the next measure
1 { changes(P,T,C) } 1 :- partTime(P,S,T-1,C-1), partTime(P,0,T,C),
                          partTimeMax(P,TM), T != TM, S == 16.
```

The rules in the preceding code fragments make clear, that a progression like in *Figure 9* with note changes for every $N_i, i \in \{1, .., 8\}$ is possible.

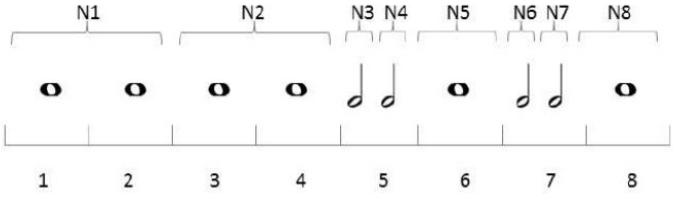


Figure 9. Armin's possible answer

The percussion instruments consist of three drum parts, the bass drum, the hi-hats and the snare drum, which all are modeled with independent files. Therefore, in *bassDrum.lp* the mandatory and optional hits, the pace and additional bass drums with lower amplitudes for more beats and nicer progressions are defined. In *hiHats.lp*, besides the hits and amplitudes, the rhythms are implemented, which can be choosen from 4 different types.

4 A Logical approach for melodic variation – alteration and musical variation with a melodic variations engine

4.1 Background of alterations in music

In music production the two main parts are composition and arrangements with varying melodic fragments. Musical variation may be regarding timbre, phrases, volume, instruments or audio extractions based on similar compositions. Variations can but doesn't necessarily need to modify one piece, as they could alter the dynamics, timbre, tempo or articulation. Here, we mainly regard the melodic alterations of a given piece and answer the problem of what notes should be preserved and what notes can vary. An alternative possibility for musical variation can be chaotic mapping, where trajectories between neighbors and the reference exist, where the extend of separation between trajectories are tempered. When speaking about musical alteration, the question arises, how much an altered piece can vary from the original, so that the essential parts remain fixed and the piece can be recognized even after multiple iterations of variations. To regard the variation degree as a computational task, parameter values can be set for the overall number of notes to be altered and a parameter value for which notes must change in a melodic line.

4.2 AIM – System Description for Melodic Lines

The System Alterations In Music (AIM) is based on Anton 2.0.0, which is a composition engine for proposing new fragments or completing a piece. Anton works by filling empty spaces of an input file with a step or a leap value depending on the previous note to complete a piece. With additional computational rules, AIM can modify single melodic lines to achieve valid alterations depending on the tonality of a given piece.

System Architecture The AIM system consists of two main sections, a logical section containing the requirements for an answer set and a sound section, which can be used for parsing the result to a musical notation with Csound. The answer set corresponds to the melodic variation from the input file. The logical section is managed by the driver file *alterDriver*, which works with parameters like the fundamental-scale, the musical mode, the bpm and the piece, to invoke the needed functions for an answer set. The input file specifies the maximum value of notes that can be changed, which then invokes a counter, sets the chosen note N regarding the counter CT with `chosenNote(P,N,CT)` and the duration with the note of the counter CT, start time ST and the duration of the note DR on measure M with `duration(ST,DR,M,CT)`. It should be noted, that the note durations are defined similarly to the Armin system, where 32 pulses correspond to a whole note, 16 for a half note and so on. One measure is 32 pulses long, which fits a 4/4 time signature. In the following fragment, an example of the defined predicates is given (*Figure 2*).

```
partTimeMax(P,5).

%%Change notes
numberOfNotesToChange(1).
toChangeNumber(1..CN) :- numberOfNotesToChange(CN).

%% choosenNote(part, note, counter).
choosenNote(1,25,1).
choosenNote(1,24,2).
choosenNote(1,22,3).
choosenNote(1,20,4).
choosenNote(1,22,5).

%% duration(start, duration, measure, counter).
duration(1,16,1,1).
duration(16,8,1,2).
duration(24,8,1,3).
duration(1,16,2,4).
duration(16,16,2,5).
```

Figure 2 Fragment of code of the input file

The melodic variations engine The main question for melodic variation is, which notes will be changed. The AIM System sets for that the value of notes that can be changed and gives few rules on the behavior of the outcome. To answer the initially asked question, AIM provides a structure for notes that should keep the essence of the piece, hence they are not changed. As the first note often indicates the fundamental, it will not be changed as well as the second note, which would keep the progression of the fundamental note. To preserve the ending of the melodic line, the last note also will not be changed. On the one hand, this works well if the melody is not too long, but on the other hand, if the melody would be longer, more notes would be changed and the essence would vary in a significant way. This system allows to perform two tasks in the melodic variation once the note to be changed is selected. Besides changing the pitch of the note, with note conversion a note can be split into two equivalent halves, in which the first note remains the same and the second one changes (*Figure 5*). If this technique is applied in multiple iterations, multiple changes and conversions are possible.

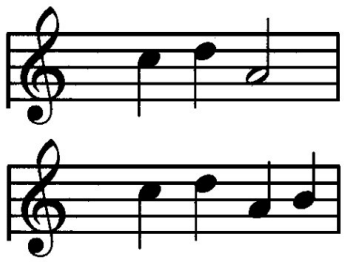


Figure 5 Note conversion. Split a note into halves

5 Conclusion

With the possibility of programmatically generating music, the question arises, how the aesthetics of music compare to rule-based generated music, like using ASP or with genetically generated music using evolutionary methods. As the results of the qualitative comparison of the aesthetics undermine, does rule-based generated music deliver already valid music regarding musical rules and aesthetic judgement with only less drawbacks in musical novelty, even compared to mutated iterations. This leads to the conclusion, that music generated with rules as constraints are still a fine option for automatic generation of music, like in the Armin System for Trance music. With Armin, the solid foundation of Anton's music generation with ASP has a noteworthy expansion regarding section chaining and a different musical genre than renaissance music. Unfortunately, I wasn't able to find any song samples of music generated by Armin, neither is the source-code publicly visible, hence the evaluation regarding musical aesthetic is not possible at the time. A main point for musical aesthetics is novelty and melodic variations, that have been introduced with the system AIM. The driver file of the logical section and the melodic variations engine are crucial for the programmatically alteration of music and offer simple methods for musical variation using ASP. Regrettably, AIM doesn't support a wide variety of application for musical alteration, as this system only supports small melodic input and gives freedom of change for nearly every tone, regardless many musical criteria. Still, this too is based on Anton and a nice extension of the basic music composition system with rhythmic and note changing. Using multiple iterations, AIM works like evolutionary musical mutation, from the point of view of aesthetics of generated music.

References

1. Georg Boenn, Martin Brain, Marina De Vos, John P. Fitch: Automatic music composition using answer set programming. TPLP 11(2-3): 397-427 (2011)
2. Andrew R. Brown: An aesthetic comparison of rule-based and genetic algorithms for generating melodies (2004)

3. Flavio Omar Everardo Pérez, Fernando Antonio Aguilera Ramírez: Armin: Automatic Trance Music Composition using Answer Set Programming. Fundamenta Informaticae 113 (2011) 79–96
4. Flavio Omar Everardo Pérez: A Logical Approach for Melodic Variations. LA-NMR (2011)