Fundamenta Informaticae 113 (2011) 79–96
DOI 10.3233/FI-2011-600
IOS Press

Armin: Automatic Trance Music Composition using Answer Set Programming

Flavio Omar Everardo Pérez*
Departamento de Computación, Electrónica y Mecantrónica
Universidad de las Américas Puebla
Sta Catarina Mártir Cholula, Puebla, México
flavio.everardopz@udlap.mx

Fernando Antonio Aguilera Ramírez
Departamento de Actuaría, Física y Matemáticas
Universidad de las Américas Puebla
Sta Catarina Mártir Cholula, Puebla, México
antonio.aguilera@udlap.mx

Abstract. The Artificial Intelligence (AI) has taken a leading role in many activities which used to be made "by hand"; one of them is the musical composition. Such task now has another alternative implementation, through the support of software that can compose different musical genres to the point where a computer can compose pieces with some autonomy. This paper proposes Armin, a system dedicated to the electronic trance music composition. Armins aim is to provide a trance music composition to serve as a template or a base audio file, ready to add more instruments in a mastering (remastering) production Additionally, it seeks to enable greater collaboration between a machine and a human, both seen as musical composers.

Keywords: Armin, Automatic Music Composition, Algorithmic Composition, Answer Set Programming, Trance Music, Artificial Intelligence

*Address for correspndence: Departamento de Computación, Electrónica y Mecantrónica, Universidad de las Américas Puebla, Sta Catarina Mártir Cholula, Puebla, México

1. Introduction

The Computer science and mathematics provide a large repertoire of algorithms that can be used to generate and process musical material [7] we mean, it is possible to automate certain musical genres based on certain knowledge obtained from its musical components and the rules inside the music style. In this paper we show that it is possible to use answer sets solvers for the composition of pieces which can vary from two to four minutes long in the trance genre. Armin is named after Armin van Buuren, a renamed trance music producer and Disc Jockey (DJ).

Armin is a system based on Anton [4] with some knowledge of the rules according to trance music, which is able to make a plan and its execution of a piece using Answer Sets Programming (ASP) [13] and at the same time assist the user in the musical production. Also Armins first version is capable of generation and executing four instruments in a single piece, three of them are percussion instruments and the other one represents a melodic line. In addition we show the models inside the trance music and the ASP application in a genre which is characterized by percussive rhythms to set the pace over time.

In the trance music, like in other music styles there are rules which describe the progression of the melodic parts and also rules which describe the overall structure of a song based on the percussion instruments. In a local level there are rules describing the behavior of a percussion instrument in a single measure.

These rules were developed to demonstrate the automation of a single music style which is based on a repetitive behavior during the song. In addition we want to deliver a proposal for composing trance music percussions with a melodic line which works as a guide of a valid progression for composing melodic and harmonic sections over this musical piece.

Also we are interest to answer in the future questions like: How much music can be automated? Is there a limit in the music automation? How can music automation be measured? Is the music automation determined by a number of instruments or if it is related to the duration of the piece? During this paper we will talk more deeply about these questions exploring the authors opinions.

2. Background Inside the Computer-aided Composition

The Computer-aided composition (CAC) [1] has taken a very important role in composition and musical production, from music pieces which have been created from synthetic instruments (Sound Synthesis) which produce lifelike sounds and a wide variety of sound effects to the automatic scores composition. All these avoid the need to go to a studio and record each instrument like before.

The CAC in the same way is known as algorithmic composition. This provides the power to map an idea to software with the options to create and edit the music contents before rendering the final song. Musicians and scientists have been studied to understand exactly how humans compose music [16] and that have raised questions as “where is the next note coming from?” [2], but the process of selecting the next note is difficult to explain and maybe it carries some personal taste. Similarly the questions arise once again. If we have already chosen a note, how long must it last? Not only are these questions which we must find the answer, also there are certain considerations to observe in detail because they may pose a problem in the CAC.

2.1. Considerations for Computer-aided Composition

One of the difficulties in the CAC is when the "power" is given completely to the tool to produce a complete piece, two cases may occur as mentioned in [16]. Create or imitate. Imitation is given to perceive a result (in this case musical) similar to any other known to us whom we can identify. What would be worth asking is: What rules allow such behavior? Similarly goes the question how much is one piece similar to another? Or how many and which notes should vary to conclude that there is a known piece by us? One solution is to find a way to get deterministic results by certain rules to provide additional knowledge and maybe some rules that crosses music boundaries.

With this proposal it is possible to explore a very large amount of results and some fragments that may lead to a "never heard before" by humans or ever even imagined by us or by the pioneers of trance music.

3. Inside the Trance Music

Trance is a genre of the electronic dance music with a percussion frequency between 130 and 140 Beats per Minute (BPM). The melodies are long and they change or evolve over time. One of the main features of trance is that its time signature is four quarters (4/4) which specifies 4 pulses (beats) per measure. Each beat is identified by a kick (Bass Drum or BD), possibly accompanied at times 2 and 4 with snare drums (SD) or clap sounds (CL).

Another feature lies in the change of pace that is usually done every two, four or eight bars. However, it is possible that given a change of pace, just add or discard drum sounds or other minor instrument. This is what is called *progression*. Throughout the song we get to hear that the rhythm changes are often strategically placed in certain places and not so random (stochastic) and it is said that the rates are increasing in intensity compared to the previous ones (also progression).

There are moments in songs where there are no beats, in other words the percussions stops to give importance to synthetic sounds, long chord and a possibly slower paces, this part of the music is known in trance as breakdowns or breaks. These breaks are made to give a stop to the beat loops, or to change in an attractive form for the rhythm of the song. The beats will later returns to resume the song's tone. A trance song structure may differ from the length of the same, but the essence remains between different versions of the song. Currently the major rankings of electronic music included in its top trance DJs as Armin van Buuren, DJ Tiësto, Markus Schulz, Andy Moor and Paul Van Dyk, to say some. Today the trance sounds have evolved and have achieved a high degree of sound perception and polyphony.

Trance music is a global style, nevertheless is also a matter of taste and varies from all across the world, but it is clear that music and now the trance music are part of today's dance life.

Creating dance or any other musical sequences of sounds depends in an important way of our cultural preferences including our musical experiences and the final product may vary from one composer to another. A guide to compose trance music in mostly of the times will start in the percussion section or beats. The bass drum composition can start by setting the four kicks per bar and claps or snare in the positions 2 and 4 of each measure. *The Dance Music Manual* suggests that the next instrument to be composed should be the hi-hats (HH) which can be "open hi-hats" or "closed hi-hats". These hats can appear in different ways among measures; they can be played every 16th time or apply a variant to this approach.

The trance music is mostly composed by loops; which is what we have described above as the percussion instruments (bass drum, snare or clap and hi-hats). These loops are tended to repeat along the song with some variations to avoid some rhythm monotony and therefore the rhythm should sound more pleasant and no repetitive and boring.

The trance music is not closed to only these three percussive instruments. It is possible to find some bongos, congas, tom drums and some other percussive instruments but an important factor in here is that trying to not saturate the loop and leave some room for melodic instruments. Adding to many percussion instruments may downplay the melodic and harmonic lines in a song.

4. Anton

Anton[1] [4] is a melodic, harmonic and rhythmic composition system which uses ASP to generate short pieces. Anton's current version is 2.0.0 and it's main tasks are:

- Compose
- Diagnosing errors
- Completing a piece.

4.1. Composition

Anton offers melodic and harmonic as well as rhythmic composition and is available for the implementation of up to four voices (solo, duet, trio and quartet). Anton works in the style of "Palestrina Rules" from Renaissance music in the modes: major, minor, Dorian, Phrygian, Lydian and Mixolydian[2].

4.2. Diagnosing Errors

This section allows detecting errors within a piece. This part also appears in the composition section for generating pieces musically valid or correct. For this part is needed to pass a file with a progression as Anton's input and depending on the fundamental and the mode, Anton will accept the progression or not.

4.3. Completing a Piece

Is possible insert a piece fragment in a logic programming format and select the complete piece option in Anton. If the part is complete Anton returns one solution or model. This model consists of a set of steps number given by the letter T which refers to the time at which the note N should be played in a part P.

The complete system consist of three major phases; building the program, running the solver and interpreting the results. Finally there is an option for playing the audio file generated by Csound [5] at the end of the execution.

[1]Official Website: http://www.cs.bath.ac.uk/ mjb/anton/

[2]For more information about these musical modes, see for example [6, 14]

5. Answer Set Programming

ASP is a declarative programming paradigm where a software program is used to describe the requirements which must be completed successfully by solving a specific problem [4], this paradigm is based on stable models (answer set). One main feature about ASP is the use of constraints; the effect of adding a constraint to a program is to eliminate some of its stable models [13] and so it is possible to avoid some undesired executions (solutions or models).

To produce musical material with this declarative approach, it is necessary to model one or more instruments in order to capture its essence to obtain musical results. To do so, each instrument both melodic and percussion consist in a set of rules which allow us to model some part of a musical piece as shown in Figure 1 and thereby determine their behavior over time. The code inside the Figure 1 is an ASP code and its main feature is that it plays the drum hits depending on the rhythm type: standard or a variation from the standard. The mandatory hits are at the times 0, 4, 8, and 12 like a common or standard trance rate (counting from 0 to 15). In addition there are other 12 possible hits in a sixteenths measure so the other 12 hits should be decided whether the hit is made or not, if the hit is done, take an amplitude value and play it. This variation only applies for the standard variation of the main pace shown in the line **1 playDrum(Y,X) : amplitude(X) 1 :- bassDrumHits(Y), type(standardVariant).** . This means that after having the standardVariant fact, select a bassDrumHit and pick minimum one and maximum one value of amplitude and set the values to playDrum. If the stroke is not carried out, it is equivalent to mention that the amplitude of the hit is zero.

```
%%mandatory BD hits for trance music
mandatoryHit(0;4;8;12).

%% if is mandatory hit... get the amplitude and play it
playDrum(X,A) :- bassDrumAmplitude(A), mandatoryHit(X).

%% Fill the optional hits with zero amp.
playDrum(Y,0) :- bassDrumHits(Y), type(standard).

%% pick an amplitude and a non mandatory hit...
1 { playDrum(Y,X) : amplitude(X) } 1 :- bassDrumHits(Y),
                                        type(standardVariant).
```

Figure 1. Fragment of Armin Bass Drum Generator Written in ASP Code

It is worth to say that the predicates can contain variables (denoted with capital letters A, X, Y) and constants (starts with lower case ”standard”, ”standardVariant”).

5.1. Why ASP?

ASP has become a popular approach to declarative problem solving in the field of Knowledge Representation and Reasoning (KRR). It supports a simple modeling language with high-performance solving processes. The main idea behind ASP is to represent a given computational problem by a declaratively logic program. The solutions to such problem are what we known as answer sets.

This approach is similarly to Satisfiability Testing (SAT) where problems are encoded and whose models represent the solutions to the given problem and ASP also is similar to Prolog in a syntactically way. In a proper manner ASP allows solving all search problems in NP (and NPNP) in a uniform way offering more than SAT [9].

The solver and grounder used for this system is Clasp [10] and Gringo [11] respectively, taken from the Potassco Project Site[3] . Both implementations are currently leading the ASP area [8] and in the algorithmic composition solving area shown in Anton [2].

6. Armin - System Description

Armin[4] is a trance music composition algorithm which takes the composition rules represented declaratively using ASP. These rules have the property to model a desired result within the genre and to explore future extended compositions. Using in part the operation of the Anton for melodic composition, Armin provides the feature to expand into the main percussion and musical sections chaining, like introduction to verse, verse to chorus, chorus to breakdown... in order to assemble these sections with some dependency for generating the next state (Markov Chains). One feature that is sought is the ability to perform a ”valid” musical composition inside the trance genre with its dependencies of the internal parts of a traditional song and percussion and its respective progression between the different sections. It should be noted that the objective of Armin in its first version is not to fully compose a piece of the trance music, but to serve as a ”template” with melodic, percussive and other sounds. Once the piece is generated, it can be use lately for some post production. Additionally it seeks to explore different outcomes (each answer a set corresponds to a valid piece of music) that have openness within the music style and proposing completely valid compositions. To generate a new piece of music just ask for a new model or answer set.

6.1. Armin - Architecture and Components

The Armin architecture is divided into two big sections of ASP as shown in Figure 2 which are the Score Generator and the Sounds Generator. It is important to add that there is a strong dependency between these two parties, because the Score’s product works as the input of the Sound Generator.

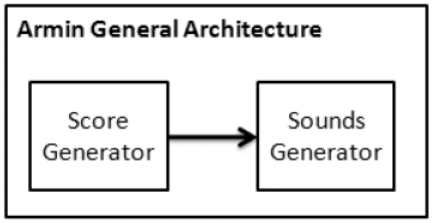

Figure 2. Armin Architecture

[3]Official Website: http://potassco.sourceforge.net/

[4]Armin official Website: Flavio Everardo: http://tranceaiproject.com and http://flavioeverardo.com in the Armin Project and Armin 1.0 Songs section.

The Score Generator consists on a driver and the assembly file. The Armin driver is in charge to manage the tasks during the score execution and is the main file which receives some input parameters, which are:

BPM valid values for a trance music composition are given inside the interval of 130 to 140 BPM including those.

Fundamental (for melodic instruments any of the 7 notes that conforms the C major scale (C, D, E, F, G, A, B).

Mode possible modes for the melodic composition such as Major, Lydian, Dorian.

Parts/Fragments this is the number of parts or sections that will be played in the piece. This value is variable depending on the length of the piece that the user wants to generate. The fragments correspond to the parts of a trance song (intro, verses, choruses, breakdowns ...).

The Score Assembler as shown in Figure 3 is an ASP file that generates the order and the appearance frequency for the parts during the resulting song. These parts, as mentioned before, correspond to the internal structure of a piece. This generation is made by certain dependencies between parts of the piece and its possible behavior over time. Each answer set corresponds to one configuration of a song (this doesn't mean that every time we get the same answer). The resulting model from the assembler file (`arminAssembler.lp`) is defined over a time T. This means that each played part corresponds to a time during the score and there is no chance that two different parts corresponds to the same T value. The code in Figure 3 shows that the intro will be played in the time 1; after that the verse, the chorus and the

```
%%The song starts with the intro
playState(intro,1) :- part(intro).

%%The verse chorus and breakDown remains to fill during the time
1 { playState(verse,T+1),
playState(chorus,T+1),
playState(breakDown,T+1) } 1 :- playState(P,T), timeScore(T),
                                statesNumber(SN), T < SN-2.

%%The song finish with the outro
playState(outro, SN) :- part(outro), statesNumber(SN).

%%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).
```

Figure 3. Fragment from the assembler file

break downs are selected from the current state to be played in the time T+1. The final state corresponds to the outro (ending of a musical composition); the number of states is given by the user as input. In addition we have to control the outcome or the structure of the piece by adding some constraints like: "two states called verse cannot be played in a time T and time T+1" and "no part can be played three times in a row". This constraint is made to assure the progression on each part; these rules allow us to get different states during the time and so the song will have some variants from one state to another.

At this point, we know that a fragment is going to be played and also we know when this is going to happen, but it has not generated any audio until now. The section in charge of this task is Sounds Generator.

Once we have obtained the score model of the song, this information works as input for the Sounds Generator section which consists of five components as shown in Figure 4, these components are: results interpretation, instruments generator (ASP), Csound parser, render, and finally the final product, an audio file.

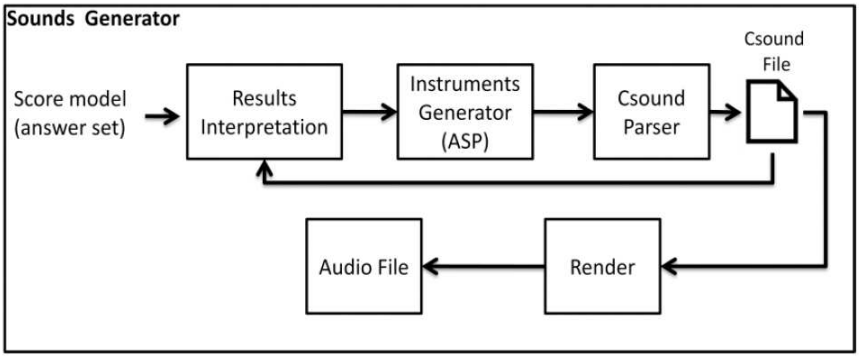

Figure 4. Sounds Generator

The Sounds Generator starts interpreting the obtained results by the model (answer set) in the Results Interpretation section. This file besides understanding the behavior of the song is responsible for sending a call to each instrument that is going to be part of the current structure by calling the Instruments Generator section. This file is responsible for generating the model of each instrument depending on the part or section in the score. Once generated the model of each instrument at a time, the Csound Parser is executed and is the responsible for mapping the results to a file in the Csound format. This process takes place until it has reached the total time of the piece or until there are no fragments remaining to be parsed. Once the file is done, it is time to run the Csound's render and then Csound provides an audio file in WAV format.

The way an automatic music composition tool creates a musical piece can vary from the goal or essence of the application, and in some ways, what it is evaluated is not the methodology but the final product (song, album, performance ...). When the application's main purpose is public and not personal [15], the methodology, internal processes and sub processes must be evaluated together with the resulting audio. If there is a personal reason or purpose behind the application, it can be said that there is neither exact methodology nor restrictions involved which limits the musical production or interpretation.

There are different approaches to the composition of a musical piece which depend on a parameter that is set as a starting or reference point for the production of the piece, such as time or duration, the

number of elements within the structure of the song or even the instrument(s) chosen. Armin presents a sequential composition structure taking the particularity that the time in Csound is cumulative, and it takes by default an entry point called introduction (intro). All pieces start off with their respective intro (this doesn't mean that all the pieces will have the same behavior) at time zero and the following states are calculated using the current state. The last fragment is dedicated to the end of the song (outro) which also has its own dependency on the previous state.

6.2. Armin - Melodic and Percussion Instruments

Armin instruments are classified into two categories; melodic and percussion instruments. The melodic instrument is set to play during the chorus and the breakdown sections in Armin's current version. This melodic instrument is based on Anton's melodic part by leaving the rules that fits in the trance style and adding some others for the performance inside the Progression and Melody files.

The melodic part is made for long behavior instruments like pads or strings and this part is made to perform eight measures. These measures have the particularity to play a whole note or two half notes except for the first measure (only plays a whole note). Once the first note is selected for the first measure, the even measures have two options: repeat the previous note (no matter if the current measure plays a whole note or halves notes) or change the value of a note N depending on the steps or leaps. When a measure is divided into two notes, the second one will always be different from the previous one to avoid the similarity to a whole note measure. An example of a possible 8 bar configurations is represented in the Figure 5.

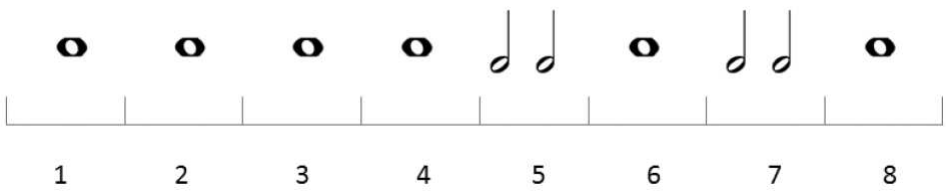

Figure 5. Example of an 8 bar configuration

For the melodic instrument we keep the same Anton's structure to create a program to get an answer set with various files: `notes.lp`, `modes.lp`, `progression.lp` and `melody.lp`. Anton 2.0 also has files for harmony (`harmony.lp`) and chords (`chord.lp`) which are not included in this first version of Armin. Also for this first version of Armin the only mode available for testing was the major mode.

The file in charge of selecting the next note is the `progression.lp` file taken from Anton and adding rules to the trance music composition. The code in Figure 6 shows some rules added with the purpose of mapping the duration of a note onto a single measure. This first version is limited to 32 pulses per measure; in other words these 32 pulses represents the duration of a whole note in a single measure. To determine the value of the whole note, we set as the minimum possible duration the thirty-second note (1/32) that means that the whole note will last the equivalent time of setting a full measure with thirty-second notes. With this classification we set that the half note will last 16. For this version of Armin we set for the melodic section these two notes (whole note and half note) but it can be expanded to a quarter, eighth, sixteenth, thirty-second and dotted notes (a half dotted and the quarter dotted note will

have a value of 24 and 12 respectively and so on) and so Armin can provide more combinations to fill a measure. The main objective of the code in Figure 6 is to fill measures with the available note durations until the answer reaches a total of 8 complete bars. It is important to mention that if the selected duration is 16, the next one will be another half note; so we can have more control of the notes inside a bar.

```
%%Select the duration of a note in a measure
pulseMeasureLimit(32).

%%Finish in the 8th measure
lastMeasure(8).

%%Duration: 16 = half note, 32 = whole note
longDurations(16;32).

%%Duration in a measure
1 { durationMeasure(0,DR,1,1) : longDurations(DR) } 1.

%% 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 6. Fragment of progression.lp

Another important fragment of code inside the progression is shown if Figure 7 which are the constraints that control the desire executions when the solver is generating the measures and their durations. The first one states that the durations in a measure cannot be greater than 32 pulses; this also works if we enable more durations to our code. The second one states that the first measure always will have a whole note inside, then the next one is a constraint that limits to two, the chances to produce measures with half notes inside and the last one says that there is no chance to get two consecutive measures with half notes. These constraints may or may not be necessary depending on the desire results by the user. To explore new outcomes just ignore them or modify them to a specific purpose.

After setting the eight measures and their respective durations, we can see in Figure 8 the way in which Armin sets the progression between the notes and measures. There are two possibilities that will lead to a trance progression, after selecting a note we can change or repeat it, here are the four cases: in many trance music progressions if the number of a measure is odd means that the note must change. The second rule states that if the measure number is even then the options are: change or repeat the note played in the last measure (odd measure). The third rule is about the measures that were divided

```
%%The chosen Duration plus the current Duration
%%must not be over the pulseMeasureLimit value
:- durationMeasure(S,DR,M,C), pulseMeasureLimit(ML), S + DR > 32.

%%the first note must last 32 pulses in the first measure
:- durationMeasure(0,DR,1,1), DR != 32.

%%The measures with half notes are limited to 2 tops.
halfNoteMeasureCounter(N) :- N = #count { durationMeasure(0,16,M,C) }.
:- halfNoteMeasureCounter(N), N > 2.#show halfNoteMeasureCounter(N).

%% Two measures with half notes cannot be at measure M and M+1
:- durationMeasure(0,16,M,_), durationMeasure(0,16,M+1,_).
```

Figure 7. Constraints inside the measure generation

into two parts, for this case the second half note will always change, no matter what note or the current measure number and the last rule states that once we got a half note played in the second part of the previous measure, the current measure must change the note. The Figure 5 is an example of an answer generated by Armin with the help of the four constraints mentioned above, another example with some notes included is shown in Figure 9.

```
%% 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.
```

Figure 8. Fragment inside progression.lp

In Figure 9 is shown a progression according to the rules mentioned before in which there are 8 notes for this configuration, but it doesn't mean that the 8 notes are completely different, that depends on the leaps and steps after a change. The measures 3, 5 and 7 changes the notes from the previous ones, also the measure 6 and 8 change the note because the previous measure is divided into two notes. In the first measure a note was selected and is also going to be played in the second measure (repeated) and the same occurs in the fourth measure after the third one change their note from being a non even measure.

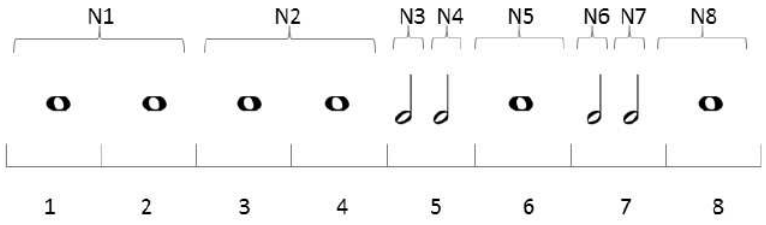

Figure 9. Armin's possible answer

In the percussion instruments we can find three drum parts: the bass drum, hi-hats and snare drum, and depending on the part of the song is the instrument or instruments that will be played in the section. The instruments are modeled with three independent files: `bassDrum.lp`, `hiHats.lp` and `snareDrum.lp`. The first one has two types of execution as shown in Figure 10; these are the "standard" and the "standard variant". The standard type is meant to produce like its name says, the basic or standard trance bass drum pace: four beats in a single measure at the times 0, 4, 8 and 12 for a 16 times bar (mandatory hits). The latter possible execution is the variation of the first one, modeling the basic pace but adding a few more bass drum hits with lower amplitude. This is made for two reasons; to add more beats and create a more "agitated" but nicer progression and to play with the amplitudes to simulate a more real bass drum hit. The reason of additionally playing a number of optional hits with a lower amplitude than the mandatory

```
%%Create the 4 measures/fragments for a trance song part (intro, verse n, chorus...).
%%All the fills are played in the fourth measure.
%%Get the models for the bass drum... mandatory bassDrumHits 0,4,8 and 12
%%for a 16 notes length

%%select if is a standard bD pace or variant
1 {type(standard), type(standardVariant) } 1.

%%optional hits
bassDrumHits(1..3;5..7;9..11;13..15).

%%volume (amplitude) values from 4 to 6 or 0 for no hit
amplitude(0;4..6).

%%default bassDrum Amplitude = 8
bassDrumAmplitude(8).

%%mandatory BD hits for trance music
mandatoryHit(0;4;8;12).

%% if is mandatory hit... get the amplitude and play it
playDrum(X,A) :- bassDrumAmplitude(A), mandatoryHit(X).
```

Figure 10. bassDrum.lp fragment code

```
%%get the models for the hi hats...
%%for a 16 notes length

%%hiHats Hits from 0 to 15 in a 16 measure
hiHatsHits(0..15).

%volume (amplitude) values from 4 to 6 or 8 for max hit amp
amplitude(0;4..6;8).

%% Select one of the four possible rhythms:
%%upTempo, downTempoAndUpTempo, all16Notes or upTempoVariant
1 { rhythm(upTempo), rhythm( upAndDownTempo ),
   rhythm(all16Notes), rhythm(upTempoVariant) } 1.

%%times 2, 6, 10 and 14 are mandatory hits in upTempo.
playDrum(2,8) :- rhythm( upTempo ).
playDrum(6,8) :- rhythm( upTempo ).
playDrum(10,8) :- rhythm( upTempo ).
playDrum(14,8) :- rhythm( upTempo ).

%%times 0, 2, 4, 6, 8, 10, 12 and 14 are mandatory hits in upTempo.
playDrum(Y,8) :- hiHatsHits(Y),
                 rhythm(upAndDownTempo), (Y mod 2) == 0.

%% All 16 times are played with different amplitudes
1{ playDrum(Y,X) : amplitude(X) } 1 :- hiHatsHits(Y), rhythm(all16Notes).

%% In the 16 hits rhythm all amps must be non zero
:- playDrum(Y,0), hiHatsHits(Y), rhythm(all16Notes).
```

Figure 11. Fragment of hiHats.lp

ones, is because in this way the listener can be conscious of the standard pace (4 beats per measure) and also can be aware whether the current measure is ending and a new one is coming with the same rhythm or with a new one.

The hi-hats are played in four different types in this version of Armin; the up tempo, up and down tempo, all 16 notes and up tempo variant. The up tempo will play hits in the positions 2, 6, 10 and 14. The up and down tempo rhythm is made to play the same hits like the up tempo rhythm besides the hits 0, 4, 8 and 12; in other words the even numbers of a 16 times measure from 0 to 15 are going to be played. Another trance rhythm inside the hi-hats is to play all the 16 notes in the measure; this rhythm is made to give certain openness to the genre. Also it has the particularity that the amplitudes can vary from one answer set to another giving us more flexibility in the rhythm. Finally the up tempo variant is made with the same purpose like the variant style in the bass drum, that give us more possible rhythms to our piece.

Unfortunately the snare drum is a percussive instrument in which we can't develop variations because the purpose of this drum is to mark the pace of the bass drum in the hits 4 and 12. The only variant we add is setting two more hits as optional to create different answers. The nature of this percussion instrument does not allow us to develop huge variants inside the trance style.

```
%%get the models for the snare...
%%for a 16 notes length

%%volume (amplitude) values from 4 to 6 or 0 for no hit
amplitude(4..6).

%%default snare Amplitude = 8
snareAmplitude(8).

%%mandatory hits 4, 12
mandatoryHit(4;12).

%%The rest of the hits are not going to be played
noHits(0..3;6..11;15).
playDrum(X,0) :-  noHits(X).

%% if is mandatory hit... get the amplitude and play it
playDrum(X,A) :- snareAmplitude(A), mandatoryHit(X).

%%optional hits 5, 13
snareHits(5;13).

%% pick an amplitude and a non mandatory hit...
1{ playDrum(Y,X) : amplitude(X) }1 :- snareHits(Y).

#hide snareHits(X).
#hide amplitude(X).
#hide snareAmplitude(X).
#hide mandatoryHit(X).
```

Figure 12. snareDrum.lp code

Armin's architecture and instruments were developed to work with the Potassco software (Clasp and Gringo) as the declarative language, the Perl programming language is used for the results interpretation and parsing and Csound for the audio synthesis.

Armin provides one command to generate a single audio file. As an example suppose we want to create a 5 fragments' piece in G major with a pace of 132 BPM:

```
arminDriver.pl --fragments=5 --fundamental=g --mode=major --bpm=132
```

7. The Automation

We believe that the music can be automated in several ways, which depends on the purpose of the software or the developers. The important question here to determine how much can music be automated is: what are the options left to the user to do after delivering a musical piece? In other words what would be the composer's contributions?

The automation cannot be measured in an easy way. Many people will only cared for the musical results and answering them if the system is doing what it is supposed to do. Perhaps in the future many algorithms composition systems will have to adopt a base or standard to be evaluated among them.

8. Evaluation and Results

Armin doesn't provide a finished work; it provides us a base (template) song, ready for adding more instruments in the way the users want. Armin delivers a musical piece as a starting point to compose trance music. The features inside a base song are the implementation of the bass drum, hi-hats, snare drum and the dependency of these instruments among the time for short or long pieces (over one minute long). In addition Armin provides openness in the drums generators, respecting the principles of the trance music, but adding some new beats features to give us different kind of songs among executions. At this point because of the length of the song it is not easy to identify the elements of a trance song by hearing, this is because the beats nuances are not completely natural. What we mentioned about the Markov Chains it is currently at a conceptual level; however is in current developing by now. We have beats fragments already in audio format which you can hear and download in Armin's site[5].

To evaluate Armin we timed this first version varying the number of the fragments from 4 to 8, with a BPM value of 135. For this results Armin was run using a 1.83 GHz Intel Centrino Duo processor, in a 32 bit version of Ubuntu Karmic Koala 9.10 with 1 GB of RAM running on a virtual machine. The same grounder and solver were used for all the executions (Gringo and Clasp) and we timed from the time when executing the command to the time where the WAV file is delivered to us.

Table 1. Time average of generated songs with a default melody line

Fragments	Running Time Average (seconds)	Duration (minutes : seconds)
4	11.1	01:56
5	12.7	02:25
6	17.5	02:54
7	22.0	03:23
8	22.8	03:52

The Table 1 shows the results of generating 10 pieces per number of fragments (average); the first column show us the number of fragments of the piece, the second, the running time and the third column is a constant value of the duration of the generated song. This is tested by setting a default melody allowing to parse only the results and not generating an answer set for the melodic part (in case the user wants to keep the melody for many tracks or to generate multiple pieces by only changing the structure of the song). In the Table 2 the melody section changes among executions; these times are an average of multiple runs but the running time depends on the number of instruments that will be executed at one time; for example one section could have only a bass drum but the next one could have the bass drum, the hi-hats, the snare and the melodic line. The results show that the system is fast enough to propose

[5]Short pieces and examples can be found in http://tranceaiproject.com in the Audios section and in http://flavioeverardo.com in the Automatic Trance Music Composition section

Table 2. Time average of generated songs. Melody lines changes for each run

Fragments	Running Time Average (seconds)	Duration (minutes : seconds)
4	13.3	01:56
5	14.0	02:25
6	19.1	02:54
7	17.7	03:23
8	24.1	03:52

a long trance piece in a short period of time. Another composition tools put their efforts on a real time composition; besides that's not our objective. We prefer to work more on the declarative rules and refine the instruments working in the musical synthesis part.

Compared to Anton 2.0.0 some rules were relaxed because we believe that these rules were not necessary for the trance music generation and that can be seen as a lower execution time. A few examples of these rules are: the piece must not finish on the fundamental; also the rests are not allowed and we allowed that a leap can cancel the previous one. Another important feature is that no matter how the structure is related to the melodic sections, a single running calculates one single model, and no matter how many times the melodic section is in the score, the only thing to do is to parse the results any time is requested.

Like Anton this system needs a musical evaluation about the generated songs. This process is a matter of personal taste but we believe that the proposed composition is acceptable for this first version; nevertheless the remaining work is still too much and is shown in the future work section. We believe that Armin is ready to explore the chances to compose extended version trance songs (over 7 minutes long).

9. Future Work

Armin is the result of combining to big areas such as electronic trance music and logical computation, and so these two areas provide us a lot of more content that can be applied here. These are reasons to believe that Armin's scope can be extended far beyond what has been seen so far, and this scope is represented two sections; Works in music and Works in computing systems.

9.1. Works in Music

Inside the musical work there are a lot of things to do starting with the addition of more instruments suitable for trance music. Also, like the Computer Systems for Expressive Music Performance (CSEMP) [12] Armin can be extended to a more real or human and not machine performance. This can be reached by working with the nuances inside the notes and beats and by adding a bass line to increase the dance movements inside the trance music.

An interesting point inside the trance music would be to model different versions of a generated piece like extended ones, which have the particularity of lasting more than six or seven minutes. Besides creating the variations of a given piece would be an interesting challenge for the taste of every user.

Another feature is the rules testing. Study the ways in which a rule is acting over the others or if it's worth to change the rule or model the rule in a different way to get another and a better meaning. Finally add more rhythms to play with and fills on every two or four measure, this will also provide the opportunity to make more "unique" pieces among executions.

9.2. Works in Computing Systems

Among the computational work we can find in a first order, the use of data mining to find more patterns inside the style and adding a plus inside the "reality" of the musical production. The second and also important will be the user's feedback in a machine learning purpose. Having the chance to "vote" or "grade" the songs quality, compositional structure, notes sequence can be a useful knowledge for future performances.

Another computational work is the chance that the user can modify and edit every step during the execution in a graphical user interface to make more suitable the human interaction.

10. Conclusions

Music is an area of personal taste; this is because we perceive music in several ways. Besides, judging is a subjective process [3] and we do not all like the same music. Armin has the flexibility of creating several solutions with the same score results and of course, it can provide more results if the score model changes among executions. This provides the users the capability to discover many unconsidered options by themselves. Adding more rules and instruments can supply us even more unexpected results and more creative pieces. Besides this allows the composers to create the basis for their own creations [2] and styles.

Acknowledgement

We would like to thank Jorge Nava who stands out as DJ and music producer, for his involvement within this paper and for their contributions and supervision within the definition of the trance music in section 3.

References

[1] Anders, T.: Composing music by composing rules: Design and usage of a generic music constraint system, 2007.

[2] Boenn, G.: Anton: Answer Set Programming in the Service of Music, *Non-Monotonic Reasoning*, Citeseer, 2008.

[3] Boenn, G., Brain, M., De Vos, M., ffitch, J.: Automatic Composition of Melodic and Harmonic Music by Answer Set Programming, in: *Logic Programming* (M. Garcia de la Banda, E. Pontelli, Eds.), vol. 5366 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, ISBN 978-3-540-89981-5, 160–174.

[4] Boenn, G., Brain, M., Vos, M., Ffitch, J.: ANTON: Composing Logic and Logic Composing, *Proceedings of the 10th International Conference on Logic Programming and Nonmonotonic Reasoning*, LPNMR '09, Springer-Verlag, Berlin, Heidelberg, 2009, ISBN 978-3-642-04237-9.

[5] Boulanger, R.: *The Csound book: perspectives in software synthesis, sound design, signal processing, and programming*, vol. 1, The MIT Press, 2000.

[6] Bulancea, G.: Mathematical models for analysing diatonic modes, *Proceedings of the 11th WSEAS international conference on Acoustics & music: theory & applications*, AMTA'10, World Scientific and Engineering Academy and Society (WSEAS), Stevens Point, Wisconsin, USA, 2010, ISBN 978-960-474-192-2.

[7] Dahlstedt, P.: Autonomous evolution of complete piano pieces and performances, *Proceedings of Music AL Workshop*, Citeseer, 2007.

[8] Denecker, M., Vennekens, J., Bond, S., Gebser, M., Truszczyński, M.: The Second Answer Set Programming Competition, *Proceedings of the 10th International Conference on Logic Programming and Nonmonotonic Reasoning*, LPNMR '09, Springer-Verlag, Berlin, Heidelberg, 2009, ISBN 978-3-642-04237-9.

[9] Gebser, M., Kaufmann, B., Kaminski, R., Ostrowski, M., Schaub, T., Schneider, M.: Potassco: The Potsdam Answer Set Solving Collection, *AI Commun.*, **24**, April 2011, 107–124, ISSN 0921-7126.

[10] Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: Conflict-driven answer set solving, *Proceedings of the 20th international joint conference on Artifical intelligence*, 2007.

[11] Gebser, M., Schaub, T., Thiele, S.: Gringo: A new grounder for answer set programming, *Logic Programming and Nonmonotonic Reasoning*, 2007, 266–271.

[12] Kirke, A., Miranda, E. R.: A survey of computer systems for expressive music performance, *ACM Comput. Surv.*, **42**, December 2009, 3:1–3:41, ISSN 0360-0300.

[13] Lifschitz, V.: What is answer set programming, *Proc. of AAAI*, 2008.

[14] Martináková-Rendeková, Z.: Searching for universal laws and rules in music, *Proceedings of the 12th WSEAS international conference on Mathematics and computers in biology, business and acoustics*, MCBANTA'11, World Scientific and Engineering Academy and Society (WSEAS), Stevens Point, Wisconsin, USA, 2011, ISBN 978-960-474-293-6.

[15] Pearce, M., Meredith, D., Wiggins, G.: Motivations and methodologies for automation of the compositional process, *Musicae Scientiae*, **6**(2), 2002, 119–148.

[16] Senaratna, N.: Automatic Music Composition Using a Tree of Interacting Emergent Systems, 2006.