ISEA2015
Proceedings of the 21st International Symposium on Electronic Art
ISSN: 2451-8611
ISBN: 978-1-910172-00-1

Automatic Genre-Dependent Composition using Answer Set Programming

Sarah Opolka and Philipp Obermeier and Torsten Schaub
University of Potsdam
Potsdam, Germany

Abstract

Harmonic music composition adheres to declarative rules and has, hence, become more and more subject to automation techniques. Specifically, Answer Set Programming (ASP), a declarative framework for problem solving, has been successfully used in recent attempts to compose music based on either a certain genre or a composing technique. However, the composition based on the combination of both has not been supported so far. This paper introduces *chasp*, an approach that considers the problem of automatic music composition from a more general perspective. More specifically, *chasp* creates simple accompanying pieces of different genres. To accomplish this ASP is used to solve the problem of chord progressions, based on the rules proposed by the theory of harmony. This results into a harmonic sequence that eventually provides the basis for the creation of simple musical pieces by applying genre-specific templates, through an additional imperative control framework.

Keywords

Automatic Composition, Answer Set Programming, Harmony theory, Logic Programming, Declarative Rule Languages

Introduction

Music creation and consumption generally involves emotions. As computers are commonly considered unable to reproduce or express human feelings, one could easily assume that it might be nearly impossible for a computer to create 'good' music. Conversely, there are many rules regarding music. Some rules regulate the way harmonies are supposed to follow one another, other rules describe the way certain composition techniques are executed (e.g. counterpoint[1] or twelve-tone technique[2]). There even exist rules for the expression of emotions through music[3]. This leads to the basic idea of using those rules to automatically compose musical pieces with a computer, which afterwards may be altered by a human to fit his needs.

Composing is mainly known as a pen and paper activity. Nonetheless, people nowadays increasingly tend to use the help of computer software which ranges from simple scorewriters like Sibelius and sequencers via music production software (e.g. MAGIX music maker[4]) and music workstations through to music creation games on the internet[5].

In-between these mostly auxiliary programs many different approaches to completely automated composition have been made and refined. (Ames 1987) and (Opolka 2012) give a detailed summary of those. Hence, only some of the most important achievements will be briefly mentioned here. One of the first known approaches to automatic composition is Mozart's "Musikalisches Würfelspiel" (Musical Dice Game)(Mozart 1793) where the player randomly chooses different snippets of music out of a given chart and plays these in a certain order, which always produces a new musical piece. Since the 1950s numerous kinds of approaches to the problem of automatic music composition emerged including Computer Aided Composition (CAC), automatic composition without any human interference, and specific new artificial languages developed for the purpose of human music composition and the application of Artificial Intelligence. In 1965, a first composition using CAC was produced by Martin Klein and Douglas Bolitho using random numbers (Ames 1987). Iannis Xenakis developed stochastic music (Xenakis 1971; Myhill 1978) and Gottfried Michael Koenig used statistics to create musical pieces (Koenig 1978; 1971). Among automatic composition there are well known works such as the Illiac Suite (Hiller 1959) or the approach on the harmonization of chorals in the style of Bach by Kemal Ebcioglu (Ebcioglu 1986; 1990). The former applied different sets of rules and Markov Chains on a given starting situation while the latter used rules of the counter point technique. Ebcioglu initially used Prolog for his approach but later he developed an own language for the problem, because Prolog was not meeting his expectations. Other languages especially developed for the purpose of composing and producing music as final output are SuperCollider by James McCartney (McCartney 1996) and ChucK by Wang and Cook (Wang and Cook 2003). A younger approach from François Pachet (Pachet 2002) utilizes Artificial Intelligence, i.e. Machine Learning. His program, Continuator, analyzes musical pieces and ap-

[1] http://en.wikipedia.org/wiki/Counterpoint
[2] http://en.wikipedia.org/wiki/Twelve-tone_technique
[3] http://en.wikipedia.org/wiki/Doctrine_of_the_affections
[4] http://www.magix.com/gb/music-maker/
[5] http://www.musicgames.com/games-by-category/compose-music-games/

plies the patterns learned in the process to create new compositions in a similar style.

chasp (Composing Harmonies with ASP), the approach introduced here, uses Answer Set Programming (ASP; (Baral 2003; Brewka et al. 2011)), a declarative reasoning framework, to produce accompanying pieces based on harmony theory and genre-specific characteristics. There already exist two similar projects using ASP, one of these being Anton, capable of creating musical compositions based on the rules of counter point (Boenn et al. 2010), the other being Armin, which is based on Anton and produces musical pieces that follow the characteristics of the trance genre (Pérez and Ramírez 2011). Although ASP is a relatively young logic programming paradigm which has not been developed for the sole purpose of composing music, it has been shown to be adequate for "teaching" the computer facts and rules about it. Both approaches mentioned earlier use rules specific to either one musical genre or composing technique and are thus only able to create music of this one small domain. Furthermore, the programs strongly depend on creating a melody and using it for the composition of the final piece. As opposed to this, *chasp* is able to autonomously create musical pieces of many different genres and doesn't even consider a melody (yet), but instead focuses on the harmonic basis of a piece. Initially, *chasp* uses ASP based on a knowledge base representing rules concerning music theory, especially harmony theory, to create a chord progression in a certain length and key which may be indicated by the user. Later, rhythm and a distinct guideline to use the notes of a given harmony are applied through a Python framework to produce different genres from one and the same harmonic basis. The decision to use ASP in this context is also motivated by its declarative nature that allows to compactly describe the problem rather than the solution. This way musical rules can easily be added or changed while at the same time it is unnecessary to consider how to execute them, as one inevitably would have to do when using an imperative approach. That is, we only use an imperative Python-based framework to transform the resulting chord progression produced by ASP into an adequate output format, i.e., as sheet music for piano in PDF and MIDI format. Altogether, *chasp* autonomously creates a musical piece based on rules concerning harmony theory and specific genres, where the user can optionally customize length, key and genre of the piece to his desire.

Initially, this paper considers the music theory behind *chasp* followed by a short introduction to ASP. Then, *chasp* is introduced in detail by first describing the process of creating chord progressions and afterwards explaining how genre specific rules are applied to these to create a musical piece. Subsequently, the paper discusses an important aspect for optimization and concludes by mentioning possible future work.

Music Theory

Music is a vast field with many different sets of known rules specific to a certain time (e.g. Baroque or modern age), an area (e.g. eastern or western) or a genre (e.g. Jazz or Metal). The approach proposed here is using western music theory based on rules from the early 19. century to the late 20th century, with a strong influence from Arnold Schönberg's "Harmonielehre" (Schönberg 1966). The general rules used by *chasp* are:

1. Two consecutive chords have to share at least one note but may never be the two same chords.
2. Disharmonies have to be prepared, i.e. the preceding chord has to contain the according disharmonic note.
3. Each cadenza begins with a tonic and ends with a dominant-tonic sequence.

The harmonic theory describes the structure of harmonies and rules for their utilization when composing with the main interest being the handling of chord progressions. There are many different types of harmonies, starting with a simple triad to which one can add one or more notes to create a chord of four or more notes. For all of these chords there exist rules defining which notes exactly form a chord as well as terms describing a specific chord's structure (e.g. Cmaj7 or IV^6).

chasp only uses two types of chords which still results in vast possibilities for different pieces of music. First, all triads from major and minor scales are being used as well as the seventh chord, where a minor seventh is added to a triad. This repertoire of chords is mostly known from folk or pop music. Still, under the consideration of rhythm and specific ways to utilize a chord's notes different genres can be created. This process will be explained in detail further below. For a more detailed explanations of the rules mentioned above as well as the resulting consequences for chord progressions you may refer to (Opolka 2012).

Answer Set Programming

ASP is a declarative framework for problem solving, originally designed for the Knowledge Representation and Reasoning domain. It is based on a simple yet expressive rule language that allows users to model problems as compact, purely declarative logic programs. The computational search for solutions, called *answer sets*, is handled by powerful high-performance ASP solvers (Gebser et al. 2007; Leone et al. 2006), whose internal design and optimization profit from the strict separation of problem description and control in the ASP workflow. Overall, ASP is a modern, proven approach to model and solve combinatorial search problems, ranging from P up to Σ_p^2-completeness, in a steadily growing number of domains e.g. Automated Planning, Robotics, Model Checking, Systems Biology, etc.

Subsequently, we only provide a brief introduction to the syntax and semantics of logic programs with cardinality rules and integrity constraints, and refer the reader to (Simons, Niemelä, and Soininen 2002) for further details. A (normal) *rule* r is an expression of the form

$$a_0 \leftarrow a_1, \ldots, a_m, {\sim}a_{m+1}, \ldots, {\sim}a_n \qquad (1)$$

where a_i, for $0 \leq m \leq n$, is an *atom* of the form $p(t_1, \ldots, t_k)$ with predicate symbol p, and $t_1, \ldots, t_k$ are terms, viz. constants, variables, or functions. Letting $head(r) = a_0$, $body(r)^+ = \{a_1, \ldots, a_m\}$, and $body(r)^- = \{a_{m+1}, \ldots, a_n\}$, we also denote r by $head(r) \leftarrow body(r)^+ \cup \{{\sim}a \mid a \in body(r)^-\}$. A (normal) *logic program* R is a set

of rules of the form (1). The *Herbrand universe* of R consists of all variable-free terms constructible from constants (by default including all integers) and function symbols occurring in R. A *ground instances* of an atom a (a rule r, resp.) occurring in R is obtained by substituting all variable in a (in r, resp.) with some element of the Herbrand universe of R. The *ground instance* (or *grounding*) of R, denoted by $grd(R)$, is the set of all ground rules constructible from rules $r \in R$. A set X of ground atoms *satisfies* a ground rule r of the form (1) if $body(r)^+ \subseteq X$ and $body(r)^- \cap X = \emptyset$ imply that $a_0 \in X$. We call X a *model* of R if X satisfies every rule $r \in grd(R)$; X is an *answer set* of R if X is a subset-minimal model of $\{head(r) \leftarrow body(r)^+ \mid r \in grd(R), body(r)^- \cap X = \emptyset\}$.

In addition, logic programs can be extended by shorthand expressions to succinctly describe certain aspects. Specifically, in the subsequent section we will make use of *cardinality rules* and *integrity constraints*. These are expressions of the form

$$h \leftarrow a_1, \ldots, a_m, \sim a_{m+1}, \ldots, \sim a_n \qquad (2)$$

where a_i, for $1 \leq m \leq n$, is an *atom* of the form $p(t_1, \ldots, t_k)$ with predicate symbol p, and $t_1, \ldots, t_k$ are terms, viz. constants, variables, or functions; the head h is either a *cardinality constraint* of the form $l\,\{h_1, \ldots, h_k\}\,u$ in which l, u are integers and $h_1, \ldots, h_k$ are atoms, or the special symbol $\bot$. We call an expression of the form (2) *cardinality rule*[6] if h is a cardinality constraint, and an *integrity constraint* if $h = \bot$. A set X of ground atoms satisfies a ground cardinality rule if $\{a_1, \ldots, a_m\} \subseteq X$ and $\{a_{m+1}, \ldots, a_n\} \cap X = \emptyset$ imply that $h = l\,\{h_1, \ldots, h_k\}\,u$ and $l \leq |\{h_1, \ldots, h_k\} \cap X| \leq u$; and a ground integrity constraint if $\{a_1, \ldots, a_m\} \not\subseteq X$ or $\{a_{m+1}, \ldots, a_n\} \cap X \neq \emptyset$. Intuitively, a ground cardinality constraint in a rule's head assures that it is only satisfied by a set of ground atoms X if X does not satisfy its body[7], or otherwise X must contain at least l and at most u atoms of $\{h_1, \ldots, h_k\}$. Furthermore, a ground integrity constraint assures that it is only satisfied by a set of ground atoms X if X does not satisfy its body. Technically, both cardinality rules and integrity constraints can be implemented as a set of normal rules, to which they are typically rewritten for uniform evaluation. Hence, the set of stable models for a logic program containing those short-hand statements is identical to the one of the normal logic program retrieved by this translation.

In the following we present our ASP programs in the technical input syntax of ASP grounder *gringo*, version 4.4 (Calimeri et al. 2012; Potassco 2014). We also assume familiarity with built-in arithmetical comparison predicates and functions typical for ASP grounders, i.e., {'=', '!=','<','<=','>=','>'} and {'+', '−', '∗', '/'}, which are evaluated upon instantiation.

[6]For simplicity, we here limit the definition to cardinality constraints occurring in heads of rules. In practice, cardinality as well as weight constraints can occur likewise in rule heads and bodies.

[7]That is, it does not hold that both $\{a_1, \ldots, a_m\} \subseteq X$ and $\{a_{m+1}, \ldots, a_n\} \cap X = \emptyset$

chasp

chasp[8] uses ASP to create chord progressions based on the rules of harmony theory and Python as imperative control to produce simple musical pieces of different genres.

Chord Progression

```
1  note(c;cis;des;d;dis;es;e;f;fis;ges;
2          g;gis;as;a;ais;b;bes).
3  next(c, ( cis; des ) ).

5  halftones(T1,T3,H+1) :-
6          note(T1); note(T2); note(T3);
7          halftones(T1,T2,H); next(T2,T3);
8          H < 12.

10 halftone_steps(maj,1,2,2).
11 key(A,K,M,S+1,T2) :-
12         key(A,K,M); key(A,K,M,S,T1);
13         halftones(T1,T2,H);
14         halftone_steps(M,S,S+1,H);
15         note_acc(A,K); note_acc(A,T1);
16         note_acc(A,T2); S < 7; H > 0.

18 triad(maj,4,3,3).
19 chord(R,T1,T2,R,maj,0) :-
20         key(A,R,maj);
21         halftones(R,T1,H1);
22         halftones(T1,T2,H2);
23         note_acc(A, (R;T1;T2));
24         triad(maj,H1,H2,_).

26 chord_inv(R,T1,T2,R,R,CM,6) :-
27         chord(R,T1,T2,_,CM,0).
```

Listing 1: Describing the musical domain

A computer has no knowledge of music. To be able to create a chord progression one first has to teach it basic musical concepts like notes, scales and chords. Only then one can describe a chord progression and rules for its creation. Listing 1 shows an excerpt of the principal rules describing the basic musical knowledge. The concept of notes is represented in lines 1 to 8. There are 12 different notes in sound and 17 in name as described by `note/1`[9] in lines 1 and 2. For each note the succeeding note is determined by `next(T1,T2)`, with `T1` being the first and `T2` the succeeding note, as shown in line 3. Between each two notes `T1` and `T2` there exists an interval of semitone steps `H` as described by `halftones(T1,T2,H)` in lines 5 to 8. Lines 10 to 16 represent the rules for scales. The intervals between any two out of seven consecutive notes produce a scale. As an example, line 10 shows `halftone_steps(maj,1,2,2)` describing the first two notes of a major scale. This can be generalized as `halftone_steps(M,T1,T2,H)`, with `M` being the mode (e.g. major), `T1` and `T2` two consecutive notes and `H` the amount of semitone steps between these two notes. The rule in lines 11 to 16 generates the actual notes of each

[8]Source code and technical documentation are available at http://potassco.sourceforge.net/labs.html

[9]`note/1` is a predicate containing one argument.

scale step. `key(A,K,M,S,T)` distinctly describes each note of a scale, with `T` being the actual note (e.g. e) on scale step `S` (e.g. 5) in key `K` and mode `M` (e.g. A Major) with `A` as the nature of its accidentals (e.g. sharps). Similar rules describe the structure of a chord in lines 18 to 24. In line 18 `triad(maj,4,3,3)` gives the intervals between all four notes of a seventh chord in a major key. Generalized, each `triad/4` describes the intervals between the notes of a specific chord. while `chord(R,T1,T2,R,maj,0)` in lines 19 to 24 generates the actual notes `R`, `T1` and `T2` of a chord with root `R` and mode `M` in no inversion. A chord's inversion is indicated by the last argument of `chord/6` which contains a number that is based on a notation known from the figured bass where `0` equals no inversion, `6` the first and `46` the second inversion. `chord_inv/7` in lines 26 and 27 gives an example for a rule generating a chord's first inversion.

```
1  next_step(scale_step_inversion(1,0),
2          scale_step_inversion(3,0)).

4  1{cadenza(N+1,T2) : next_step(T1,T2)}1 :-
5          cadenza(N,T1); cadenza_step(N+2).

7  cadenza_chord(N+1,R,CM,I) :-
8          cadenza(N+1,(S,I2));
9          cadenza_chord(N,_,_,_);
10         chord(R,T1,T2,T3,CM,I);
11         cadenza_step(N+1);
12         thiskey(K,M); key(_,K,M,S,R);
13         key(_,K,M,_,T1); key(_,K,M,_,T2);
14         key(_,K,M,_,T3); indicator(I,I2).

16 cadenza_notes(S,R,T1,T2,T3) :-
17         cadenza_chord(S,R,M1,I);
18         chord(R,T1,T2,T3,M1,I);
19         thiskey(K,M); key(_,K,M).
```

Listing 2: Creation of a cadenza

With this knowledge base one can describe the desired solution, i.e. the chord progression. The essential part are the rules describing chord progressions themselves by determining which chords may follow one another and which may not. For each chord this is represented by rules like the one shown in lines 1 and 2 of Listing 2. Each `next_step/2` describes a valid chord progression. The cardinality rule defining `cadenza/2` in lines 4 and 5 generates a chord progression by choosing exactly one succeeding chord from all eligible `next_step/2`. A chord is represented as `scale_step_inversion(S,I)` with `S` being the scale step over which the chord is established and `I` being its inversion. Again, the notation is based on the figured bass. The representation used here implies, that the lowest note of a chord in a given inversion is the note of the scale step over which the chord is being build, e.g. in the key C Major `scale_step_inversion(6,6)` represents a chord with the lowest note a in its first inversion[10], which applies to F Major. As chords are normally represented in the latter notation, lines 7 to 14 translate `cadenza/2` into that form, which is described by `cadenza_chord/4`. For further use of the chord progression (shown in the next section) a third representation containing the actual notes of each chord is provided by `cadenza_notes/5` in lines 16 to 19.

[10]The lowest note in the first inversion is the third of a basic triad.

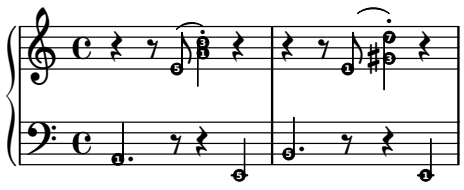

Figure 1: Sample for an accompaniment in the style of a tango

The solution output from ASP contains `cadenza/2`, `cadenza_chord/4` and `cadenza_notes/5` atoms. `cadenza/2` contains a lot of information compressed into an abstract notation based on numbers. This notation is not exactly human-readable but instead designed for ASP to be able to compute a valid next chord through mostly arithmetic calculations. To provide a faster and easier understanding of a created cadenza, the additional representation `cadenza_chord/4` is generated and integrated into either the raw ASP output when debugging or the final sheet music to give an overview of the harmonic structure of each created chord progression. As our Python framework does not contain any background knowledge about music theory and, thus, cannot directly deduce a chord's notes from its chord representation, these are provided by ASP through `cadenza_notes/5`. The process of sheet music creation through the Python framework will be explained in the following section.

Musical Piece Creation

Based on (Frank 1996), (Frank 1997) and (Kroepel 1977) general characteristics of different genres have been extracted to create lilypond templates. Using Python these are applied to the chord progression given by ASP to create musical pieces. This process will now be explained by means of an example.

(Frank 1996) proposes the two measures given in Figure 1 as a sample for an accompanying pattern in the style of a tango. A lilypond template describing these rules would contain the exact same rhythm and notes at the corresponding times. These notes would not be absolute but rather be described by a number which represents the scale step of each note in the key of the chord per measure. Translating these rules to a lilypond notation with numbers instead of explicit notes results in the following template for the right hand's first measure:

```
r4 r8 5,( <1 3>4\staccato) r4
```

This template can now be applied to the notes generated by ASP to create a complete and valid lilypond file. A cadenza generated by ASP contains information on the notes played at one time step in `cadenza_notes(T,R,N1,N2,N3)` as shown in Listing 2 with `T` being time or step of the chord in

the cadenza, `R` the root of the chord, `N1` the third, `N2` the fifth and `N3` depending on the nature of the chord either the seventh or the root again. Considering one time step equals one measure the resulting tango piece consists of as many measures as the given cadenza contains chords. As an example, when generating a four chord cadenza in e-minor ASP may give the following output:

```
thiskey(e,min),
cadenza(1,(1,0)), cadenza(2,(4,7)),
cadenza(3,(5,7)), cadenza(4,(1,0)),
cadenza_chord(1,e,min,0),
cadenza_chord(2,a,min,7),
cadenza_chord(3,b,min,7),
cadenza_chord(4,e,min,0),
cadenza_notes(1,e,g,b,e),
cadenza_notes(2,a,c,e,g),
cadenza_notes(3,b,d,fis,a),
cadenza_notes(4,e,g,b,e)
```

`thiskey/2` contains the key in which the cadenza was created. The details of `cadenza/2`, `cadenza_chord/4` and `cadenza_notes/5` were introduced in the previous section. This output is passed to Python and used to create the actual sheet music by filling out a lilypond template and thus creating a complete lilypond file. With the actual notes of the first measure as in `cadenza_notes(1,e,g,b,e)`, a complete lilypond notation for the right hand's first measure would thus become:

```
r4 r8 b,( <e g>4\staccato) r4
```

Eventually, we create a complete lilypond file by applying the lilypond template to the output given by ASP resulting in sheet music as shown in Figure 2.

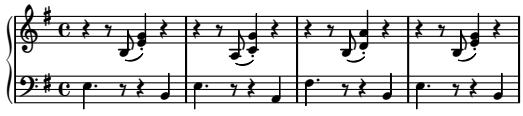

Figure 2: Final sheet music output created with lilypond

Optimization

The implementation shown so far has one weak point: with our current ASP program it takes unreasonably long to find solutions for cadenzas with a length of more than five chords. The cause for this is a relatively large grounding of the program due to its large number of rules for allowed chord progressions. In fact there are more than 600 of these particular rules. A common way to optimize the performance of an ASP program is the reduction of possible choices by abstraction. Therefore, the music theory behind the program was analyzed once more to discover a way to abstract chord progressions: instead of an extra rule for each and every valid chord progression, all chords were put in one of three groups to which the same rules for succeeding chords could be applied. The first group contains all triads in no or the first inversion, the second group all second inversions and the third group contains all seventh chords in all their inversions. These groups are represented by rules like `group/2` in the lines 1 to 3 of Listing 3 which describes group 1 as explained above. Lines 5 to 9 show the new version of `next_step/2` which we introduced in Listing 2. Now the rule describes valid progressions for groups of chords instead of a single specific chord. For a detailed explanation of these formulas you may refer to (Opolka 2012).

```
1 2 {group((T,0),   (T,          0));
2    group((T,0), (((T+1)\7)+1,  6))}2  :-
3         scale_step_inversion((T,0)).

5 next_step((T1,0), (T2,46)) :-
6   group((T1,0),_); group((T2,46),_);
7   1 { T2 =  T1   \7+1;
8       T2 = (T1+3)\7+1;
9       T2 = (T1+5)\7+1 }.
```

Listing 3: Abstract rules for chord progressions

To illustrate the differences in the implementation's performance before and after, Table 1 shows a comparison of resource-related statistical data for the problem instance of generating a 5 chord cadenza in C Major, which was run with gringo 4.4.0 and clasp 2.1.4 on an Intel(R) Celeron(R) CPU 1017U @ 1.60GHz with 4 GB RAM. Furthermore, we modified the experiment by increasing the number of chords to 6 which yielded no results for the naive encoding after running for 30 mins, whereas the abstract encoding was even capable to find a 50 chord cadenza within 5.5 seconds.

statistics		naive encoding	abstract encoding
Time	:	6.350s	3.004s
CPU Time	:	6.300s	2.970s
Choices	:	121	40
Conflicts	:	12	4
Atoms	:	15928	11120
Rules	:	36559	19628
Equivalences	:	33676	23098
Variables	:	8253	3001
Constraints	:	37463	13380

Table 1: Excerpt from clasp's statistics output when generating a 5 chord cadenza

Future Works and Conclusion

In this work we introduced a method to create musical pieces of many different genres on the basis of one chord progression. With minimal (and not neccessarily required) input through the user *chasp* autonomously generates music based on rules regarding harmony theory and genres.

Particularly, the declarative nature of our approach makes it convenient to add and change rules to describe the underlying concepts of composition. An obvious future enhancement for *chasp* lies, thus, in the addition of further genres through additional lilypond templates. It might be worth considering the possibility to convert part of the information used to create these templates into rules for ASP. Currently, the possibility to create a 4-voices homophonic choral is in development. Integrating additional types of chords and their specific rules in

our approach poses an increasing challenge. So does enhancing the current result by adding a melody to the accompaniments as well as rhythm for the choral output. Besides that, it may also be worth considering to add the possibility to reverse the current work flow and create an accompaniment to an already existing melody.

Acknowledgments This work was partially funded by DFG grant SCHA 550/9-1. We are grateful to the anonymous reviewers for their suggestions.

Authors Biographies

Philipp Obermeier is a research assistant in the Knowledge-Based Systems group at University of Potsdam, Germany. His scientific interest is centered around Answer Set Programming and the development of declarative languages for dynamic domains.

Sarah Opolka received the M.Ed. for the main fields Music and Computer Science from the University of Potsdam in 2015. Her master thesis on *chasp* is based on her bachelor thesis on automatic composition.

References

AMES, C. 1987. Automated composition in retrospect: 1956-1986. In *Leonardo Vol. 20, No. 2, Special Issue: Visual Art, Sound, Music and Technology*, 169–185.

BARAL, C. 2003. Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press.

BOENN, G.; BRAIN, M.; DE VOS, M. and FFITCH, J. 2010. Automatic Music Composition using Answer Set Programming. In *Theory and Practice of Logic Programming*.

BREWKA, G.; EITER, T. and TRUSZCZYŃSKI, M. 2011. Answer set programming at a glance. *Communications of the ACM 54 (12)*, 92–103

CALIMERI, F.; FABER, W.; GEBSER, M.; IANNI, G.; KAMINSKI, R.; KRENNWALLNER, T.; LEONE, N.; RICCA, F. and SCHAUB. T 2012. ASP-Core-2: Input language format. Available at https://www.mat.unical.it/aspcomp2013/files/ASP-CORE-2.0.pdf

EBICOGLU, K. 1986. An expert system for chorales harmonization. In *Proceedings of the Fifth National Conference on Artificial Intelligence (AAAI-86)*, 784–788.

EBICOGLU, K. 1990. An expert system for harmonizing chorales in the style of J. S. Bach. In *Journal of Logic Programming Vol. 8, No. 1*, 145–185.

FRANK, B 1996. Rhythm-Styles for Piano, Vol. 1. Schott.

FRANK, B 1997. Rhythm-Styles for Piano, Vol. 2. Schott.

GEBSER, M.; KAUFMANN, B.; NEUMANN, A. and SCHAUB, T. 2007. clasp: A Conflict-Driven Answer Set Solver. In *Proceedings of the Ninth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'07)*, 136-148.

HILLER, L. A. and ISAACSON , L. M. 1959. Experimental music. Composition with an electronic computer. In *Journal of Music Theory Vol. 3, No. 2*, 302–306. Duke University Press, on behalf of the Yale University Department of Music.

KOENIG, G. M. 1971. The use of computer programs in creating music. In *Music and Technology (Proceedings of the Stockholm Meeting organized by UNESCO)*. Paris: La Revue Musicale. 93-115. http://www.koenigproject.nl/Computer_in_Creating_Music.pdf

KOENIG, G. M. 1978. Programmed music. http://www.koenigproject.nl/Programmed_Music.pdf

KROEPEL, B. 1977 Piano Rhythm Patterns. Mel Bay Publications, Inc.

LEONE, N.; PFEIFER, G.; FABER, W.; EITER, T.; GOTTLOB, G.; PERRI, S. and SCARCELLO, F. 2006. The DLV System for Knowledge Representation and Reasoning. *ACM Trans. Comput. Logic Vol. 7, No. 3*, 499–562.

MCCARTNEY, J. 1996. Supercollider: a new real time synthesis language. In *Proceedings of the 1996 International Computer Music Conference*, 257–258

MOZART W. A. 1793. Anleitung so viel Walzer oder Schleifer mit zwei Würfeln zu componiren so viel man will ohne musikalisch zu seyn noch etwas von der Composition zu verstehen. KV Anh. 294d, J. J. Hummel, Berlin-Amsterdam.

MYHILL, J. 1978. Some simplifications and improvements in the stochastic music program. In *Proceedings of the 1978 International Computer Music Conference*.

OPOLKA S. 2012. Theoretische Grundlagen der automatischen Komposition. Bachelor Thesis, University of Potsdam.

PACHET, F. 2002. Playing with virtual musicians: the continuator in practice. In *IEEE Multimedia, Vol. 3*, 77–82.

PÉREZ, F. O. E. and RAMÍREZ, F. A. A. 2011. Armin: Automatic Trance Music Composition using Answer Set Programming. In *Fundamenta Informaticae Vol. 113*, 79–96.

POTASSCO, 2014. Potassco website. Available at http://potassco.sourceforge.net

SCHÖNBERG, A. 1966. Harmonielehre. Universal Edition, Wien.

SIMONS, P.; NIEMELÄ, I.; and SOININEN, T. 2002. Extending and implementing the stable model semantics. *Artificial Intelligence* 138(1-2):181-234.

WANG, G. and COOK , P. R. 2003. Chuck: A concurrent, on-the-fly, audio programming language. In *Proceedings of the 2003 International Computer Music Conference*.

XENAKIS, I. 1971. Formalized music: Thought and mathematics in composition. Pendragon Press.