UNIVERSIDADE DA CORUÑA

Faculty of Computer Science

Technical Report

Haspie: A Musical Harmonization Tool powered by Answer Set Programming

Authors: Martín Prieto, Rodrigo
Cabalar Fernández, José Pedro

A Coruña, April 7, 2017.

Abstract

Haspie is a musical harmonization and composition assistant based on Problem Solving and Constraint Logic through the use of Answer Set Programming. The tool takes scores in MusicXML format and annotates them with the best possible harmonization. If specified, it is also able to complete intentionally blank sections and create brand new parts of the score. At its current state, the tool is mostly functional, but has limitations. Despite this fact, it serves as a Proof of Concept of one of the many real world applications that Answer Set Programming can achieve to solve.

Keywords:

- √ Logic Programming
- √ Knowledge Representation
- √ Answer Set Programming
- √ Constraint Solving with Preferences
- √ Computer Music
- √ Harmony
- √ MusicXML

Contents

List of Figures

Chapter 1
Introduction

Music Theory learning has been stuck in the same old-fashioned methods and systems for years. These methods are based on exercises and repetition, meant to train the ear and become fluent in composition or in solving the proposed exercises. Harmony learning is one of the most important steps in music, since being able to analyse scores in this way is vital for its comprehension, later interpretation and further development. Haspie aims to help Harmony students achieve a better understanding of the matter and lets them experiment earlier with composition from an harmonic point of view.

Constraint Logic fits well with the problem since the harmony rule set used in the most basic levels taight in music schools hasn't changed since the origins of the subject, it's a definite set of rules, not very big and quite strict. Simply translating this rule set to ASP constraints and being able to extract the fatcs and knowledge from any score, the tool is able to detect any mistake and propose solutions, as well as filling in any blank sections of the score to create, in the end the harmony of the piece.

The great advantage that Answer Set Programming provides is that there is no need to create nor optimise a solution searching algorithm, since it only needs these harmony rules. Answer Set Programming not only offers this simplicity and power, but also flexibility since a change in the rules or in the optimization configuration can lead to very different results and adapt better to the composition style of the user.

Chapter 2
Background

2.1 State of the Art

Most of the research done in computer music is extracted from the relationship between musical theory and mathematics. This relationship is fairly easy to understand and model in a mathematic way.

There are two main lines of research in computer music but hey often overlap since the research matter is the same, despite changing the point of view. These lines are Composition Assistance Tools and Composing-Oriented Intelligent Systems. Haspie lies in a point between these two lines of research: It's a tool to assist the composition but it's able to understand and develop any score given to it.

2.1.1 Answer Set Programming

The core of the project is *Answer Set Programming* [1] (ASP from now on). The main module of the project has been developed through the use of the Stable Models of Gelfond & Lifschitz [2] and non-monotonic logic. ASP is a declarative programming language oriented to solve hard search problems, mostly NP-complex ones.

Since it's a declarative language, ASP programs only requires the inference rules and constraints of the problem to solve any instance of it, these instances need to be previously translated to ASP facts. ASP uses a grounder to expand these facts using the inference rules of the program to calculate the solutions that will be later pruned through the previously mentioned constraints, thus finding the solution (or multiple solutions if there were more than a single one). This methodology is known as *generate and test*.

The ASP Tools developed by the Potassco Group[1] include, among others, a *grounder* (Gringo) and a *solver* (Clasp), but these tools are packed into a single program called Clingo, that does both grounding and solving.

2.1.2 MusicXML

The format that haspie uses as input for the musical scores is MusicXML. MusicXML, MXML or *Music Extensible Markup Language* is an extension of the XML format used to represent occidental music. No only it does include the score information but also includes how it should be represented on paper (margins, font sizes, musical notes position coordinates in the sheet, etc.)

It uses xustom XML tags to group different levels of musical information together such as the general piece data, parts, measures and such. (See Figure 2.1). It's a very rich format but due to it's complexity and very easy parsing properties it's meant for computer use more than for human writing and reading.

```
<note default-x="74.65" default-y="-25.00">
        <pitch>
                <step>A</step>
                <octave>4</octave>
        </pitch>
        <duration>1</duration>
        <voice>1</voice>
        <type>quarter</type>
        <stem>up</stem>
</note>
```

Figure 2.1: Sample A note represented in MusicXML

[1]http://potassco.sourceforge.net/

2.1.3 Software

2.1.3.1 Tools and Libraries

- **Flex and Bison** are Unix utilities that allow the creation of fast text parsing tools.
- **Music21** is a suite of tools that helps students and musicians alike to query info about well-known musical pieces. Not only it includes a very complete musical database but it also includes libraries to help in programmatic music composition.

2.1.3.2 Musescore2

Musescore2 is the editor of choice to work along haspie. It's open source and free and allows to import and export all of the common music representation formats, even synthesizing the score sound in MIDI and other sound formats.

2.1.3.3 Intelligent Systems

In the AI field there has been many approaches to the musical composition subject form many different points of view of this field.

- **EMI and Emily Howell** Developed by David Cope [3], EMI or Experiments in Music Composition, is a system capable of identifying the style of a musical piece and complete any required section of it following the same style. Cope's work studies the possibility of using grammars along with dictionaries in musical composition. After the work on EMI, it derived into the software known ad Emily Howell. Emily Howell uses EMI to create and update it's musical database but has it's own interface through which the user can give feedback and modify the current composition. Cope polished Emily Howell with his own musical style to create and compose many music albums that where later published.
- **ANTON** [4] is a rhytmic, melodic and armonic composition system based in Answer Set Programing. ANTON is capable of composing small musical pieces from scratch or parting from a given score. It uses the renaissance

style of Giovanni Pierluigi da Palestrina since it's a well defined set of musical composition rules that can be translated to ASP rules.

- **Vox Populi** [5] uses genetic algorithms to compose music in real time. This system starts from a population of chords codified throught the MIDI protocol to then perform the required genetic steps (crossings, fittings, mutations, etc) selecting the best of them each iteration through pure physical methods regarding sound frequencies.

- **CHORAL** is an expert system that works as a harmonizer with the classic style of Johann Sebastian Bach. The rules used by the system represent the musical knowledge from the different points of view of the choral group. To achieve the harmonization, the program uses a generate and test with backtracking algorithm.

- **CHASP** is a tiny tool created by the Potassco Group to calculate chord progressions through Answer Set Programming starting from scratch, allowing the user to specify key and length of the piece. Despite not having an input file, it allows exporting the result and post-processing it to imprint different rhythmic styles to the composed piece.

Chapter 3

Method

3.1 Approach

The architecture of `haspie` (See Figure 3.1) is a simple pipeline wirtten in Python with a single execution path. This pipeline calls each submodule passing the correct parameters and then retrieves and parses the output generated by each step of the tool.

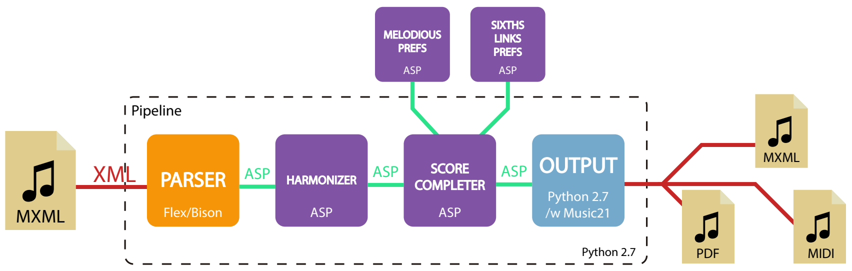


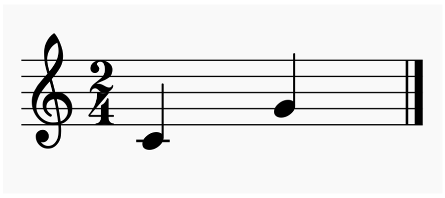
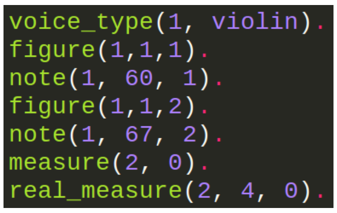
Figure 3.1: Haspie's Architecture

3.1.1 Input

The tool takes a single MusicXML file as input that is passed to the first stage of the pipeline: a parser written in C along with the Flex and Bison libraries. This module transforms the MusicXML tag information to Answer Set Programming logic facts. This parser does not only parse the musical figures but also performs many other tasks such as:

- Subdividing the figures to standardize the rhythmic patterns.
- Creating additional logic facts to retrieve the original rhythm structures after the score processing.
- Identifying the different measure types.
- Interpreting the instrument names to determine their most common pitch ranges.
- Infer the tonality of the score by parsing it's key.
- Parsing some of the score metadata, such as the piece name or the composer.

After the parsing step, all of this extracted data is saved in the form of configuration and answer set programming files. (See Figure 3.2)

```
voice_type(1, violin).
figure(1,1,1).
note(1, 60, 1).
figure(1,1,2).
note(1, 67, 2).
measure(2, 0).
real_measure(2, 4, 0).
```

Figure 3.2: Small piece parsed to ASP facts

3.1.2 ASP Core

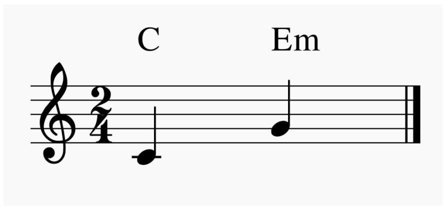
The previously created ASP fact file is the direct input of the harmonization finder step, which is the first half of haspie's ASP Core. These module uses these facts to expand the general harmony rules, thus inferring support hidden predicates and finally assigning a chord to each specified section of the piece.

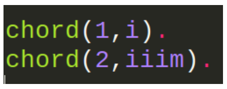
```
1 { chord(HT,C) : pos_chord(C) } 1 :- htime(HT).
```

The possible chords are defined in separate files `major_chords` and `minor_chords`, the tool determines which to use by inferring the mode from the extracted key. These chords are defined by the tonal relation between their notes and not by their notes particular name. By doing it this way, the tool generalizes the chord concept, reducing it to a tonal grade detection and then fitting the best possible chord for that grade by taking all the notes in the analyzed beat interval. To do so, the parsed notes' pitches are abstracted to tonal grades of the corresponding scale (using the inferred key). This is achieved by performing simple operations to the note's semitone value.

```
octave(V,((N - base) / 12),T) :- note(V,N,T), N >= 0.
sem_tones(V,((N - base) \ 12),T) :- note(V,N,T), N >= 0.
grade(V,1,T) :- sem_tones(V,3,T).
grade(V,2,T) :- sem_tones(V,5,T).
grade(V,3,T) :- sem_tones(V,7,T).
grade(V,4,T) :- sem_tones(V,8,T).
grade(V,5,T) :- sem_tones(V,10,T).
grade(V,6,T) :- sem_tones(V,0,T).
grade(V,7,T) :- sem_tones(V,2,T).
```

Knowing the tonal grades of each note of the rhythmic interval, the tool then marks the notes in strong beats as mistakes and, by the use of optimization rules, searches the answer that minimizes the amount of mistakes





```
chord(1,i).
chord(2,iiim).
```

Figure 3.3: Chords annotated over a small piece and their corresponding output logic facts

Back to the pipeline, the best possible found harmonies are parsed to an object Python representation that makes possible their future representation and music score recomposition. The tool displays a summary of these best answers and lets the user choose one for the completion step, by default the tool uses the solution with least mistakes in it's harmonization. A temporal chord facts file is then created, that is used, along with the original logic facts to complete the blank parts or the new voices of the score. (See Figure 3.3)

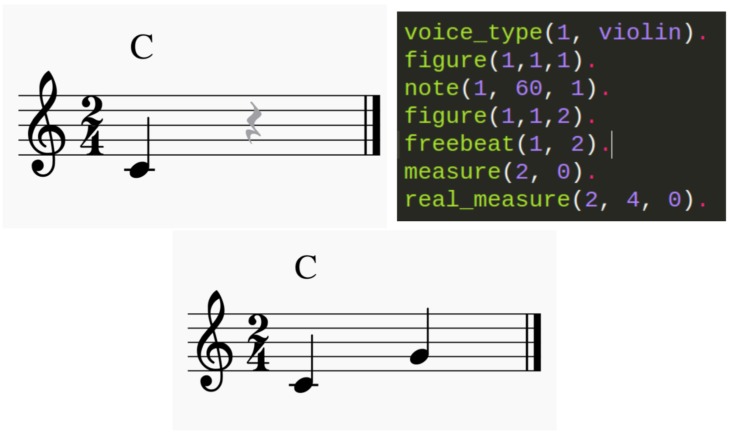
The second half of the tool is called if there are blank sections in the score or the creation of new parts were specified to the pipeline. This second half works in a similar fashion as the first half, by assigning new notes to the completable sections of the score among the pitch range of the specified instrument or voice type of the part.

```
1 { freebeatfigure(V,N,1,FB) : N=VL..VH } 1 :- freebeat(V,FB),
                    voice_limit_low(V,VL), voice_limit_high(V,VH).
```

These new notes are again generalized to their tonal grade and octave (as it's done in the very first steps of the previous half) to then being checked against the selected harmonization. This is done by marking the incorrect ones in strong beats as mistakes, as well as checking for other melodic rules such as note distance or trying to avoid certain undesirable sounds produced among the different voices that play at the same time. By minimizing these mistakes, again, the optimal solutions are found.

```
octave_jump(V,B1,B2) :- ex_note(V,N1,B1), ex_note(V,N2,B2),
                        (B1+1) == B2, N2 > (N1+12), beat(B1+1).
octave_jump(V,B1,B2) :- ex_note(V,N1,B1), ex_note(V,N2,B2),
                        (B1+1) == B2, N2 < (N1-12), beat(B1+1).
:- octave_jump(_,_,_).
```

Finally, thanks to the extra rhythmical predicates extracted by the parser, the original piece is reconstructed.



```
voice_type(1, violin).
figure(1,1,1).
note(1, 60, 1).
figure(1,1,2).
freebeat(1, 2).
measure(2, 0).
real_measure(2, 4, 0).
```

Figure 3.4: Completed harmonized score with an incomplete beat

Repeating the previous process, the user chooses a complete score solution and then the pipeline stores the new information in object structures for their final file output in the specified format. (See Figure 3.4)

3.1.3 Preferences Modules

Haspie has two optional preferences modules that add some further rules to improve the second half of the ASP core results.

The first one is a melodic preference module. As haspie does not have a well-defined composition rule set, this module aims to smoothen the completed blank sections of the score. It has rules for:

- Defining the melody tendency
- Shortening the melodic jumps between notes

For tendency, the rules infer if a section has a rising or falling tendency and tries to imitate that tendency in the completable sections. For the melodic jump smoothening, new predicates infer these jump sizes and optimization rules search for the solutions that minify these jumps.

```
melodic_jump(V,J,B1,B2) :- out_note(V,N1,B1), out_note(V,N2,B2),
                          (B1+1) == B2, beat(B1+1), J = #abs(N1-N2).
```

The second module detects certain popular choral chord progressions (fourth and sixth inversions) and try to complete them by selecting the correct notes for them in the completable blank sections. These module performs a second per-beat harmonization, making it very slow computationally.

Haspie contains a standard configuration files that weight both these preference modules and the both halves of the core optimization values. By changing these values, the harmonization and composition results may be altered, making it easier for the user to change the way the tool works without having to implement or modify new ASP rules or preference modules.

3.1.4 Output

The last module called by the pipeline uses the score's internal representation Python objects to export the result in the format specified by the user. This module works using the music21 library [1] developed by the MIT. The internal structure is translated to this library own object representation to then being exported easily to any of the supported formats.

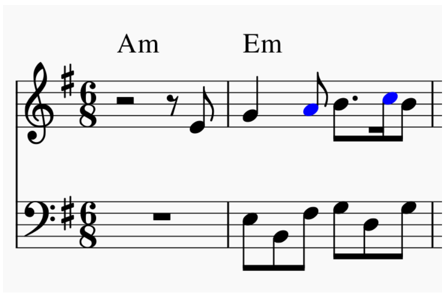


Figure 3.5: Sample harmonized score tih passing notes marked in blue

[1]http://web.mit.edu/music21/

Chapter 4
Evaluation

En este capítulo exponen los resultados de la evaluación del sistema. Se han seleccionado tres piezas musicales conocidas para realizar las diferentes pruebas. A continuación se describen las tres piezas y las pruebas realizadas. Para realizar una prueba de carga que establezca los límites del sistema se ha añadido una última partitura suficientemente sencilla sobre la que trabajar para poder realizar estas pruebas cómodamente sin preocuparse por la calidad de los resultados.

For the tool's evaluation, three pieces were chosen to test the different aspects of the tool. To perform a workload test of the tool, a last simple piece was added to these three first pieces, just to measure execution times, regardless of the quality of the results.

- **Minuet in Major G:** Famous Johann Sebastian Bach musical piece. Interesting to check the tool's performance in ternary measures.
- **Greensleves:** By Henry the VIII, it's a complex four-part polyphony, useful to check the tool's harmonization performance.
- **Joy to the World:** Well known Georg F. Händel jingle.
- **Twinkle Twinkle Little Star:** Very simple piece to test workload and measure run times of heavy computation tasks.

Using each of these pieces, the tool was asked to harmonize and complete a measure of each as well as a whole new part, measuring not only runtime but

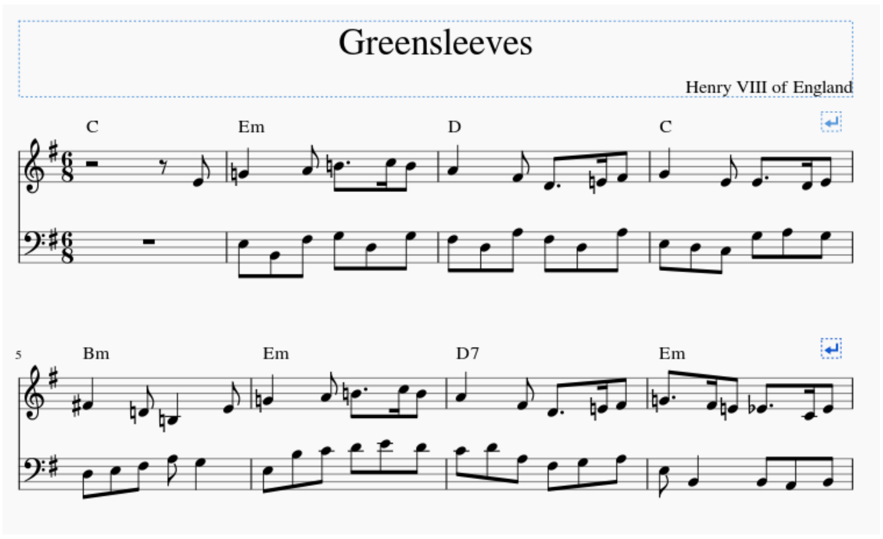


Figure 4.1: Harmonization result of the first measures of Greensleeves

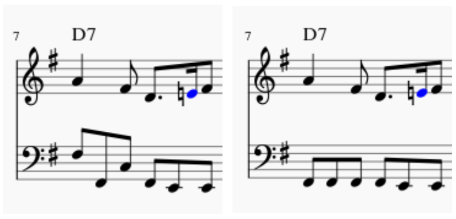


Figure 4.2: Completed single measure of Greensleeves with and without melodic preferences

also quality of the result. Due to the non-deterministic nature of ASP, as well as the I/O times, the time measures are just to provide an approximate idea of the run times of the tool. Each measure was taken 100 times, subtracted user input time to each and then averaged.

Piece	Harmonization	Measure	New Part
Greensleeves	1.016s	1.926s	4m 49.032s
Menuet	0.631s	0.726s	3m 50.376s
Joy to the World	2.381s	3.813s	7m 17.115s
Twinkle Twinkle	0.685s	0.716s	2m 31.299s

The run time results of the tool are the expected for an ASP tool. Harmony selection times are very good and the completion times are very promising. Nevertheless, the required time to complete new parts grows very quickly as adding more and more sections to complete makes the possibilities grow exponentially.

In quality terms, the selected chords are correct, and the section completion or the new parts creation offer interesting harmonically correct solutions.

The load tests for “Twinkle Twinkle Little Star” were performed by emptying progressively more and more measures of one of the parts. The piece has 24 measures, and were emptied in blocks of four. Finally more and more voices were added to the piece, achieving the following run times.

Test	Time
4 compases	1.481s
8 compases	2.394s
12 compases	3.978s
16 compases	3.982s
20 compases	5.966s
1 voz	2m 31.299s
2 voces	25m 17.298s

4.1 Comparison

Haspie is quite unique in what it does and how does it achieve it. The only comparable tool would be ANTON [4] as both use ASP to perform musical harmonization and composition and both can work over new or previously created scores.

ANTON is a way more complex system, allowing not only harmonic but melodic composition and also features rhythmic patterning, thing that haspie lacks.

Haspie is a bit more deep in harmony terms, allowing any style of musical piece to be processed, when ANTON only works with pieces of one or two voices of a certain musical style.

The tests ran for ANTON for pieces of two parts and 32 measures have run times in the minutes order, while haspie achieves a similar performance when creating a second part but it’s still hard to compare, as haspie does not work in the complexity level that ANTON does.

4.2 Known Issues

The tests performed revealed a few recurrent problems. In the Future Work section 4.3 it's detailed which of them will be addressed in short or mid term.

- **Triplets:** Triplets and similar irregular figures don't work properly and need to be edited in the input score before processing it. This is because of the rhythmic standardization that the parser performs, being unable to assign correctly the time of these kind of figures.
- **Key:** Some of the inferred keys are mis-interpreted in the output. This is a visual mistake more than a functional one and produces no further problems.
- **Voice Names:** The locale of the system affects the parsing of the voices' names, producing some mis-interpretations.
- **Falsos positivos:** Due to the simple implementation of the identification of strong and weak beats, some notes marked as mistakes or passing notes in the score may not be so.

4.3 Future Work

The main guideline of Future Work is about developing a user friendly UI and correcting visual output errors as well as expanding the project to achieve one of the restrictions initially imposed to it, such as modulation.

4.3.1 Aesthetic and GUI

With the release of improved versions of the music21 library, the visual errors of the score output should be fixed. Regardless of this, there is the intention of implementing the tool as a Musescore2 plugin so the user could be able to harmonize and complete scores live, instead of having to switch back and forth between the score editor and the command line.

4.3.2 Parsing and Harmonization

The key detection should be improved, as well as the parts name and their voice types. The strong and weak beat detection needs further work, to make it able to detect these kind of beats in complex rhythmical patterns, thus improving the harmony selection and completion results. This point also includes complex figures such as the mentioned triplets.

4.3.3 Modulation

Modulation was one of the self imposed restrictions of the project to keep it's development time in check. Not only is hard to parse and detect but also presents problems to the tool due to it's way of detecting the best chords and the way it completes blank sections. The main approach for this should be splitting the score in sub-scores, preforming the current steps and finally re-assembling it.

4.3.4 Publishing

After making the tool more user friendly and fixing some of the tool's limitation, professionals of the music teaching sector should be contacted to provide feedback and to further improve the tool so it can finally fulfill it's role as a full-fledged music learning tool.

Appendix A
Installation

In it's current state, it's advised to install the tool in a Linux environment. This is mostly because of the use of Flex and Bison libraries, that are easily compiled and used in that kind of environments.

The requirements are Python 2.7 and the latest versions of Flex and Bison (as well as a C compiler). Other than that, gringo 3.0.5 and clingo 3.0.5 are needed, those can be downloaded at the Potassco Group's Sourceforge page [6] as well as the music21 module in it's latest version. All these tools and libraries should be accessible on the System's `PATH`

It is advised to install any tools required to compile or visualize the desired output formats before installing music21 so it can be properly attached to them without any further configuration.

Of course a score edition tool is also advised, it is completely optional but it's convenient. Musescore2 was used for the tests and demos as it's Open Source and free, but any other similar editor that can handle MusicXML files will work.

The tool can finally be executed by calling it's main entry point on a command line, and any of the parameters as well as it's usage explanation can be checked with the `-h` or `--help` options.

Bibliography

[1] G. Brewka, T. Eiter, and M. Truszczyński, "Answer set programming at a glance," *Commun. ACM*, vol. 54, no. 12, pp. 92–103, Dec. 2011. [Online]. Available: http://doi.acm.org/10.1145/2043174.2043195

[2] M. Gelfond and V. Lifschitz, "The stable model semantics for logic programming." MIT Press, 1988, pp. 1070–1080.

[3] D. Cope, *Experiments in Musical Intelligence*, 2nd ed. A-R Editions, 1996.

[4] G. Boenn, M. Brain, M. De Vos, and J. ffitch, "Automatic Music Composition using Answer Set Programming," *ArXiv e-prints*, Jun. 2010.

[5] A. Moroni, J. Manzolli, and R. Gudwin, "Vox populi: An interactive evolutionary system for algorithmic music composition," *Leonardo Music Journal*, vol. 10, pp. 49–45, 2000.

[6] "Potassco - the potsdam answer set solving collection," accessed: 2016-01-08. [Online]. Available: http://potassco.sourceforge.net/

[7] "music21 a toolkit for computer-aided musicology," accessed: 2015-11-05. [Online]. Available: http://web.mit.edu/music21/

[8] E. Herrera, *Teoría Musical Y Armonía Moderna - Volumen 1*, 1st ed. Antoni Bosch, 2015.

[9] MIDI Manufacturers Association, "Midi 1.0 detailed specification."

[10] G. Boenn, M. Brain, M. De Vos, and J. Ffitch, "Automatic music composition using answer set programming," *Theory and Practice of Logic Programming*, vol. 11, no. 2-3, pp. 397–427, 2011.

[11] G. H. Knud Jeppesen, *Counterpoint: the polyphonic vocal style of the sixteenth century*, 1st ed. Englewood Cliffs, N. J., Prentice-Hall, 1990.

[12] V. Tran, "Music composition using artificial intelligence," 2009.

[13] G. Brewka, T. Eiter, and M. Truszczyński, "Answer set programming at a glance," *Communications of the ACM*, vol. 54, no. 12, pp. 92–103, 2011.

[14] V. Marek and M. Truszczyński, "Stable models and an alternative logic programming paradigm," in *The Logic Programming Paradigm*, ser. Artificial Intelligence, K. Apt, V. Marek, M. Truszczynski, and D. Warren, Eds. Springer Berlin Heidelberg, 1999, pp. 375–398. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-60085-2_17

[15] I. Niemelä, "Logic programs with stable model semantics as a constraint programming paradigm," *Annals of Mathematics and Artificial Intelligence*, vol. 25, no. 3-4, pp. 241–273, 1999. [Online]. Available: http://dx.doi.org/10.1023/A:3A1018930122475

[16] "LilyPond: Musical notation for everyone," accessed: 2015-11-05. [Online]. Available: http://www.lilypond.org