haspie - A Musical Harmonisation Tool based on ASP[⋆]

Pedro Cabalar and Rodrigo Martín

Department of Computer Science,
University of Corunna, Spain.
{cabalar,r.martin1}@udc.es

Abstract. In this paper we describe a musical harmonisation and composition assistant based on Answer Set Programming (ASP). The tool takes scores in MusicXML format and annotates them with a preferred harmonisation. If specified, it is also able to complete intentionally blank sections and create new parts of the score that fit with the proposed harmonisation. Both the harmonisation and the completion of blank parts can be seen as constraint satisfaction problems that are encoded in ASP. Although the tool is a preliminary prototype still being improved, its basic functionality already helps to illustrate the appropriateness of ASP for musical knowledge representation, which provides a high degree of flexibility thanks to its relational, declarative orientation and an efficient computation of preferred solutions.

1 Introduction

Music Theory learning is a field with a long tradition, relying on well-established methods but suffering, in many cases, from an obsolete technology. A central discipline in music learning is *Harmony*, a cornerstone for score analysis, vital for its comprehension, later interpretation and further development. Harmony studies the superposition of sounds, that is, the combination of simultaneous notes to create *chords*. In their turn, chord progressions help either to reaffirm or to blur (depending on the author's intentions) the concept of *tonality*. It is usually said that harmony constitutes the "vertical" reading of a score (that is, what is sounding at a same time) as opposed to the "horizontal" reading provided by the *melody* (that is, how a given voice progresses along time).

Apart from acquiring theoretical foundations, Harmony students must usually face a series of exercises in the form of a four-voices score partially incomplete. The goals of these exercises may comprise tasks such as annotating the chords, filling missing voice parts or both. This has to be done according to some style *rules* that may be strict or, occasionally, act as preferences or recommendations. In other words, harmony exercises constitute a natural application domain

[⋆] This work has been partially supported by MINECO, Spain (project TIN2013-42149-P), Xunta de Galicia, Spain (projects GPC ED431B 2016/035 and 2016-2019 ED431G/01 for CITIC center) and European Regional Development Fund (ERDF).

for constraint based reasoning. Moreover, musical notation (at least for harmony purposes) is mostly discrete and symbolic, and its knowledge involves many kinds of relations (among notes, chords, measures, etc), becoming in this way an ideal test-bed for logical Knowledge Representation and Reasoning (KRR).

In this paper, we present `haspie`, a musical harmonisation tool based on *Answer Set Programming* (ASP) [1–3], one of the most successful KRR paradigms for practical problem solving. `haspie` aims to help Harmony students achieve a better understanding of the matter and lets them experiment earlier with composition from a harmonic point of view. The tool can be combined with a score graphic editor (`MuseScore`[1]) to create/manipulate harmony exercises that can be solved via a call to the ASP backend. The solutions can then be displayed again in the graphic editor or directly translated into MIDI files for their reproduction.

2 Tool description

The architecture of `haspie` (Fig. 1) is a simple pipeline written in Python with a single execution path. The input format for `haspie` is *Music Extensible Markup Language* (MusicXML, or MXML), an extension of the XML format used to represent Western music. It contains not only score information but also display data such as margins, font sizes, musical notes position coordinates in the sheet, etc. The tool takes a single MusicXML file (usually generated with the graphic editor `MuseScore`) 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 ASP facts. The parser also performs other tasks such as fixing the measure level at which the harmonisation will take place, interpreting the instrument or voice names to determine their most common pitch ranges, reading the expected tonality via the key signature, etc.

The core of `haspie` is an ASP encoding that declares the style norms and preferences as logic program rules. Without entering into detail (see [3] for a extended explanation), ASP usually describes constraint problems with a methodology called *generate-define-test*. Some rules allow generating potential solutions, constraints (test) are used to prune undesired choices and an additional group of rules allow defining auxiliary predicates to represent some features used in generation and test. An example of a generation rule is:

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

It essentially asserts that, for each time beat `HT`, exactly 1 chord `C` is (non-deterministically) assigned among all those possible chords `pos_chord(C)` defined elsewhere according to the chord harmony rules. With this rule alone, each possible assignment becomes one of the answer sets of the program, which are in one-to-one correspondence to solutions to our harmonisation exercises.

As we can see, ASP rules allow variables (in capital letters) that are replaced by their possible instances in a first stage called *grounding*. After all rules are

[1] `https://musescore.org/`

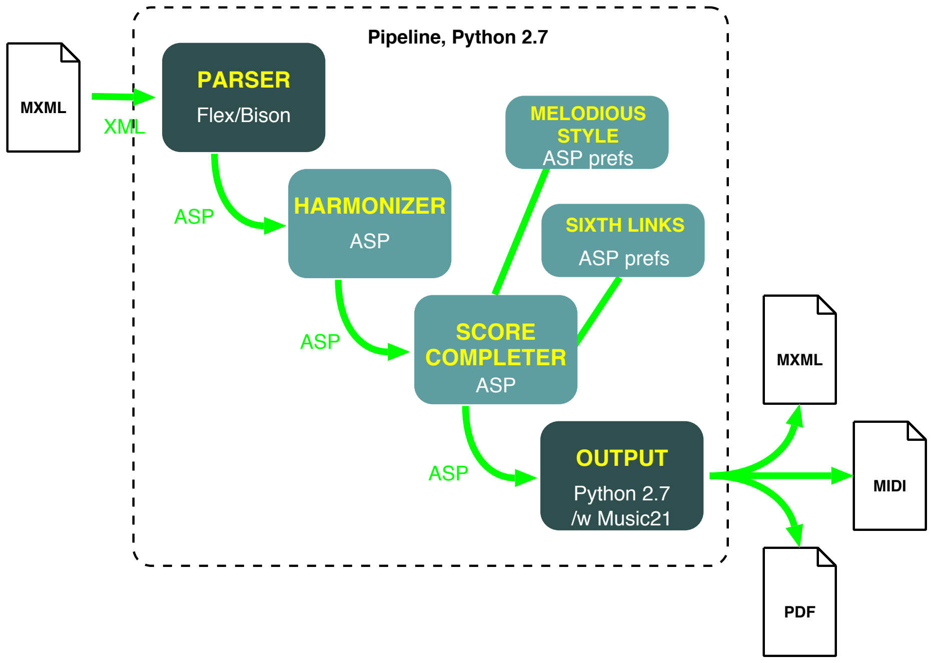

Fig. 1. `haspie` Architecture

grounded, a second stage does the (propositional) solving. The ASP solver that `haspie` uses as a backend is `clingo`[2] which incorporates both the grounding and solving phases in a singe package. To reduce the combinatorial explosion, the ASP core for `haspie` has been actually divided into two modules: the harmonisation module and the score completer (see Figure 1). The former assigns a (coherent) harmonic structure for the partial score provided as an input, whereas the latter fills the gaps once the harmony has been established.

The harmonisation module takes a file with ASP facts and uses this information to expand the general harmony rules, using some auxiliary predicates and finally assigning a chord to each specified section of the piece with the rule shown before. The possible chords `poss_chord(C)` are defined in separate files `major_chords` and `minor_chords` – the tool determines which one to use by inferring the mode from the extracted key signature. These chords are defined by the relative role of the notes in the (inferred) tone scale (1st grade, 2nd grade, etc) rather than their absolute pitch (C, D, etc). By doing so, the tool generalises the chord concept, reducing it to a tonal grade detection and then fits the best possible chord for that grade by taking all the notes in the analysed beat interval. Knowing the tonal grades of each note of the rhythmic interval, the tool then marks as mistakes those notes in strong beats not belonging to the assigned chord so that, using optimisation rules, answer sets with a minimum number of

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

mistakes are chosen. The tool displays a summary of these best answers and lets the user choose one for the score completion step A temporal chord facts file is then created, that is used, along with the original logical facts to complete the blank parts or the new voices of the score.

The second half of the tool, the score completer, is called if there are blank sections (denoted using predicate `freebeat`) in the score or the creation of new parts were specified in the input options. This second half works in a similar fashion as the first half, by assigning new notes to the completable sections of the score among those in the pitch range of the specified instrument or voice type of the part. For instance, the rule:

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

is a non-deterministic assignment of a note `N` between the lowest `VL` and highest `VH` pitches for voice `V`, for each time point `FB` marked as `freebeat` (that is, blank section to be filled). These new notes `N` are again generalised to their tonal grade and octave (as it is done in the very first steps of the previous half) to then being checked against the selected harmonisation. 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.

The following is an example showing a hard constraint:

```
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(_,_,_).
```

The first two rules define the auxiliary predicate `octave_jump(V,B1,B2)` that detects when a given voice `V` moves from one note at beat `B1` to another note at `B1+1` with the same name but one octave higher (or lower). Rules like the one in the last line, without a head expression before '`:-`', correspond to constraints in ASP. In this case, the constraints rule out all octave jumps.

`haspie` has an optional module including some preferences to improve the result of the score completer. These preferences include some melodic rules trying, for instance, to minimise the size of melodic 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).
#minimize[melodic_jump(_,J,_,_) = (J * weight) @ priority].
```

The `minimize` clause above assigns a penalty `J * weight` per each melodic jump of `J` semitones. The constant `weight` and the `priority` level for this preference can be assigned by the user, so we can tune how much influence this preference will have on the final result. The preferences module includes other optimisations. For instance, if a section melody has a rising or falling tendency, it tries to imitate that tendency in the completable sections as much as possible. Another group of preferences detects a popular type of chord progression (cadential 6/4 chords), trying to extend a sequence of this type of chords as much as possible by filling completable blank sections. The last module called by the pipeline uses the internal score representation in Python objects to export the result in the format specified by the user. This module works using the `music21` library[3] developed by the MIT.

3 Evaluation

For the tool evaluation, four pieces were chosen. For each one, the tool was asked to harmonise and complete one of its measures as well as a whole new part, measuring not only runtime but also quality of the result. The left table in Fig. 2 shows the obtained results: each measure was taken 100 times and then averaged. Harmony selection times are very good and the completion times are very

Piece	Harmonisation	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

Test	Time
4 measures	1.481s
8 measures	2.394s
12 measures	3.978s
16 measures	3.982s
20 measures	5.966s
1 voice	2m 31.299s
2 voices	25m 17.298s

Fig. 2. Some execution results.

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 like the one in Fig. 3, adding an harmonically correct bass line to the well-known melody from "Greensleeves." The load tests for "Twinkle Twinkle" were performed by leaving blank measures in one of the parts. Measures were emptied in blocks of four. We also tried adding 1 and 2 voices to the piece, achieving the times in the right table of Fig. 2.

4 Conclusions and Related work

Clearly, the closest works to `haspie` are `ANTON` [4] and `CHASP` [5], since both rely on ASP too. `ANTON` is a complex rythmic, melodic and harmonic composer for

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

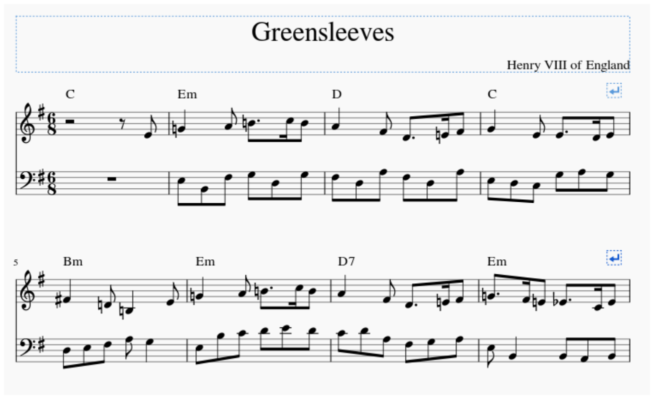

Fig. 3. Obtained harmonisation and bass part for Greensleeves melody.

small musical pieces. Although it is more complete than `haspie`, it is limited to the Renaissance style of Palestrina and only two voices, so it is less suitable for harmony training. `CHASP` is a tiny tool created by the Potassco Group to calculate chord progressions through ASP starting from scratch (no input file), allowing the user to specify key and length of the piece.

The current prototype is only at a preliminary stage. Many open topics remain for future work. Apart from improving the input and output interfaces (like the connection to `MuseScore` as a plugin) a fundamental extension which is a current, important limitation, is the possibility of automated *modulation*, that is, allowing `haspie` to decide changes in the piece tonality in a dynamic way.

References

1. Niemelä, I.: Logic programs with stable model semantics as a constraint programming paradigm. Annals of Mathematics and Artificial Intelligence **25**(3-4) (1999) 241–273
2. Marek, V.W., Truszczyński, M.: Stable models and an alternative logic programming paradigm. In Apt, K.R., Marek, V.W., Truszczyński, M., Warren, D.S., eds.: The Logic Programming Paradigm. Artificial Intelligence. Springer Berlin Heidelberg (1999) 375–398
3. Brewka, G., Eiter, T., Truszczyński, M.: Answer set programming at a glance. Communications of the ACM **54**(12) (December 2011) 92–103
4. Boenn, G., Brain, M., De Vos, M., Ffitch, J.: Automatic music composition using answer set programming. Theory and Practice of Logic Programming **11**(2-3) (2011) 397–427
5. Opolka, S., Obermeier, P., Schaub, T.: Automatic genre-dependent composition using answer set programming. In: Proc. of the 21st Intl. Symposium on Electronic Art (ISEA'15). (2015)