MusAssist: A Domain Specific Language for Music Notation

Ilana Shapiro
Pomona College
issa2018@mymail.pomona.edu

ABSTRACT

MusAssist is an external, declarative domain specific language for music notation that bridges the abstraction gap between music theory and composition. Users can describe complex musical templates for all triads and seventh chords, all diatonic as well as chromatic and whole tone scales, the five primary cadences (including imperfect authentic), and the four primary harmonic sequences with desired length. Uniquely, the level of abstraction of a template MusAssist matches that of the theoretical musical structure it describes (e.g. users can specify a harmonic sequence without manually lowering the granularity to chords and notes). Thus, users can write out specifications precisely at the conceptual levels of the musical theoretical structures they would organically conceive when composing by hand. In MusAssist, users can also change key signatures, start a new measure, and describe fundamental musical objects such as notes, rests, and chords comprised of custom collections of notes. A musical expression described by a higher level template is expanded out (i.e. the level of abstraction is fully lowered to notes) by the Haskell-based MusAssist compiler and is finally translated to MusicXML, a language accepted by most major music notation software, for further manual editing.

1. INTRODUCTION

When writing music, composers must manually transition from musical theoretical concepts to notes on a page. This process can be tedious and slow, requiring the composer to expand by hand complex structures, such as cadences and sequences, to the notes that they constitute. The level of abstraction of the musical theoretical structure is higher than what the composer actually writes.

Domain specific languages, or DSLs, are programming languages highly specialized for a specific application and thus characterized by limited expressiveness. An *external DSL* has custom syntax that is separated from the primary language of its application. MusAssist is an external, declarative DSL for music notation that bridges the divide between music theory and notation. Users describe a composition in MusAssist's straightforward, high-level syntax, modeled around the musical elements composers organically conceive when notating by hand, and the MusAssist compiler writes out the music via these instructions. MusAssist's declarative programming paradigm was chosen to further correspond with the declarative nature of handwritten music.

Fundamentally, MusAssist supports notes (including rests) and custom chords (i.e. any desired collection of notes) in the octave and key of choice, as well as commands to change the key signature or start a new measure. MusAssist is unique in that users can also write specifications for complex musical templates *at the same level of abstraction as the musical theoretical structures they describe*. MusAssist supports templates for **chords** and **arpeggios** (all triads and seventh chords in any inversion), **scales** (all diatonic scales, as well as chromatic and whole tone), **cadences** (perfect authentic, imperfect authentic, plagal, half, deceptive), and **harmonic sequences** (ascending fifths, descending fifths, ascending 5-6, descending 5-6) of a desired length. The musical expression described by a specification is expanded (i.e. the abstraction level is fully lowered to notes) by the Haskell-based MusAssist compiler.

The target language of the MusAssist compiler is MusicXML, itself a DSL that is an extension of XML (Extensible Markup Language). MusicXML is accepted by most major notation software programs (such as MuseScore). Thus users can open can open the resulting MusicXML file of a compiled MusAssist composition in MuseScore or another program for further customization and editing, thus bypassing the need to write out complex musical templates by hand at a note- and chord-level of abstraction. Beyond a professional music compositional aid, MusAssist may be particularly helpful to music theory students as an educational tool, enabling them to visualize the relationship between a theoretical musical structure and its expanded form, such as a cadence and the chords resulting from its expansion.

2. RELATED WORK

The era of music DSLs began in 2008 with Ge Wang's ChucK audio processing language, which spans the application domains of "methods for sound synthesis, physical modeling of real-time world artifacts and spaces (e.g., musical instruments, environmental sounds), analysis and information retrieval of sound and music, to mapping and crafting of new controllers and interfaces (both software and physical) for music, algorithmic/generative processes for automated or semi-automatic composition and accompaniment, [and] real-time music performance." Since then, researchers have taken advantage of the increased flexibility afforded to DSLs via their limited expressiveness to create music DSLs tailored towards notation, algorithmic composition, signal processing, live coding with music performance, and more. In the notation domain, Mu-

sicXML, LilyPond, and PyTabs stand out.

Michael Good's MusicXML is an Internet-friendly, XML-based, declarative DSL capable of representing Western music notation and sheet music since c. 1600. It acts as an "interchange format for applications in music notation, music analysis, music information retrieval, and musical performance," thus supporting sharing between specialized applications.

MusicXML attempts to emulate for online sheet music and music software what the popular MIDI format did for electronic instruments. It is derived from XML in order to help solve the music interchange problem: to create a standardized method to represent complex, structured data in order to support smooth interchange between "musical notation, performance, analysis, and retrieval applications." XML has the desired qualities of "straightforward usability over the Internet, ease of creating documents, and human readability" that translate directly into the musical domain, and it is more powerful and expressive than MIDI.

MusicXML is more expressive than MusAssist, but the level of abstraction of all musical elements is extremely low (i.e. chords must be written out as individual notes) and its syntax is very difficult and tedious to write by hand. However, its flexibility and expressiveness make MusicXML an excellent target compilation language for MusAssist's user-friendly syntax and high-level musical theoretical templates.

LilyPond, an external declarative DSL created by Han-Wen Nienhuys and Jan Nieuwenhuizen, is similar to MusAssist. It features a "modular, extensible and programmable compiler" written in Scheme to generate Western music notation of excellent quality and supports the mixing of text and music elements. Text-based *musical expressions*, or fragments of music with set durations, are compiled to an aesthetically formatted score.

LilyPond and MusAssist are both music notation DSLs tailored to non-programming audiences. However, they differ in two fundamental areas: (1) MusAssist supports complex music templates at the levels of abstraction of the musical structures they represent, whereas LilyPond only supports more granular, low level composition of individual notes and chords, and (2) the output of the MusAssist compiler is intentionally editable via notation software, unlike LilyPond's compiler, which produces a static, printable PostScript or PDF file by taking in a file with a formal representation of the desired music.

Simic et al.'s external, declarative DSL PyTabs similarly is geared toward music notation, but in a different domain than MusAssist. Specifically, the authors attempt to solve visual problem of tablature notation, and the lack of standardization of how to specify note duration in this format, by consolidating these issues into a formal language. Tablature notation is outside the scope of MusAssist's focus on Western musical theoretical structures.

3. LANGUAGE FEATURES

3.1 Low-Level Fundamentals

On the most basic level, MusAssist supports individual rests and notes. Rests are given a duration from sixteenth to whole note, and notes are further defined by note name (A to G), accidental (double flat to double sharp), and octave (1 to 8, after the range of a piano). Just as in normal notation, the absence of an accidental indicates natural quality. Finally, users can also define "custom chords," or user-defined lists of individual notes. These are not considered templates as the high-level description of the chord is not given, and the granularity is at the note level.

3.2 High-Level Templates

MusAssist supports templates for chords, arpeggios, scales, cadences, and harmonic sequences, specified uniquely at the abstraction level of the musical theoretical structures they represent.

Precisely as in music theory, chords are specified by the root note (defined as the fundamental MusAssist note is), quality (major, minor, augmented, diminished, or half diminished), inversion (root, first, second, or third), and chord type (triad or seventh). Half diminished and third inversion options apply to seventh chords only. The root note cannot have a double accidental, as this can introduce triple accidentals in the chord, which MusAssist does not support.

Arpeggios are defined precisely as MusAssist chords are, as arpeggios are simply broken chords.

Similarly, diatonic scales are given by key and scale type (major, and harmonic/melodic/natural minor), while non-diatonic scales are simply specified by their type (chromatic or whole tone). A scale is either ascending or descending, must be given a length, and does not necessarily begin on the tonic – the start note must be supplied (just as the fundamental MusAssist note is). Following notation convention, chromatic scales are notated with sharps when ascending and flats when descending.

Cadences are specified by cadence type (perfect or imperfect authentic, half, plagal, or deceptive) and key (defined by a fundamental MusAssist note and a quality, either major or minor). Currently, MusAssist only supports a single treble clef line. Thus, cadences are written out in the upper voices only, in keyboard voice leading style and incorporating principles of smooth voice leading.

Based on the principles of functional harmony, there are several ways to represent a cadence. In MusAssist, the following representations were chosen. The major version is presented first, and the minor after that, in parentheses (Table 1).

Perfect Authentic	IV-V-I (iv-V-i)
Imperfect Authentic	IV-vii$^{o\,6}_{4}$-I$^{6}_{4}$ (iv-vii$^{o\,6}_{4}$-i$^{6}_{4}$)
Plagal	IV$^{6}_{4}$-I (iv$^{6}_{4}$-I)
Deceptive	IV-V$^{6}_{4}$-vi$^{6}_{4}$ (iv-V$^{6}_{4}$-VI$^{6}_{4}$)
Half	IV-ii^{6}-V (iv-ii^{o6}-V)

Table 1: MusAssist Cadences Summary

All cadences except perfect authentic are built exclusively with triads. Perfect authentic cadences also double the root in the final chord to simulate the 4-5-1 bass line as well as to preserve the requisite 2-1 downward step in the uppermost voice. This is demonstrated in Figure 1, produced with the MusAssist syntax

```
(Perfect Authentic Cadence, Eb5 minor, sixteenth)
```

compiled and loaded into MuseScore notation software.

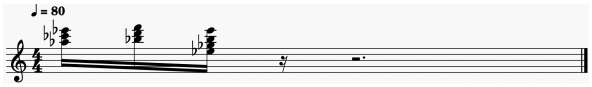

Figure 1: Perfect Authentic Cadence in E♭ minor

Finally, harmonic sequences are specified by harmonic sequence type (ascending fifths, descending fifths, ascending 5-6, descending 5-6), key (just as in cadences), duration of each chord, and length of the sequence. Since MusAssist does not yet support multi-line composition, harmonic sequences are written like cadences in keyboard-style voice leading. Though the upper-voice harmonization of a harmonic sequence need not follow the direction in the sequence's name, MusAssist chooses a chord inversion and voice leading pattern such that each sequence does align with the direction of its name (i.e. ascending-named sequences will ascend directionally) and also maximizes smooth voice leading.

In music theory, harmonic sequences can be implemented in several ways depending on desired inversion scheme. The chosen chord progressions for each MusAssist sequence are summarized in Table 2. All sequences are shown in major for the sake of example, but their inverse in minor is equally supported. Each consists of fourteen distinct chords before repeating in the subsequent octave.

Ascending Fifths	I^6_4	V	ii^6_4	vi	iii^6_4
	vii^o	IV^6_4	I	V^6_4	ii
	vi^6_4	iii	vii^{o6}_4	IV	
Descending Fifths	I	IV^6_4	vii^o	iii^6_4	vi
	ii^6_4	V	I^6_4	IV	vii^{o6}_4
	iii	vi^6_4	ii	V^6_4	
Ascending 5-6	I	vi^6	ii	vii^{o6}	iii
	I^6	IV	ii^6	V	iii^6
	vi	IV^6	vii^o	V^6	
Descending 5-6	I^6_4	V	vi^6_4	iii	IV^6_4
	I	ii^6_4	vi	vii^{o6}_4	IV
	V^6_4	ii	iii^6_4	vii^o	

Table 2: MusAssist Harmonic Sequences Summary

3.3 Additional Features

Beyond compositional elements, users can set the key signature at the beginning of any measure up to seven sharps or flats by specifying note name, accidental, and quality (sharp or flat). Users can also start a new measure or create a blank measure. Finally, users can label MusAssist expressions and reuse them later in the program. MusAssist comments are designated with \\.

The tempo for all MusAssist programs is set at ♩ = 80bpm and cannot currently be customized or changed. This also apples to the time signature, which is set at $\frac{4}{4}$.

All compiled MusAssist programs adhere to standard notation conventions. Notes and rests are broken over barlines as well as over the strong beat (beat three) of the measure. They are divided greedily into valid rhythmic units (i.e. from sixteenth to whole note) ordered either least to greatest, or greatest to least in the case of spillage over the barline into the following measure.

4. SAMPLE PROGRAM

The full breadth of MusAssist's syntax is demonstrated in Figure 2, and Figure 3 is the result of opening the compiled MusicXML code from Figure 2 in MuseScore.

```
SET_KEY A major
SET_KEY A major
SET_KEY G major
(D4 whole) (F#4 quarter) (Ab4 dotted_quarter) (G#4 eighth) (rest sixteenth)
// note without b or # is natural
notes1 = (D4 whole) (F#4 quarter) (Ab4 quarter) (G#4 eighth) (rest whole)
chords1 = ([Bbb5, Db5, C5] half) ([C#5, E5] half)
chord = (D6 minor arpeggio, root inversion, eighth)
(F#4 half diminished seventh chord, second inversion, eighth)
(D4 whole) (F#4 dotted_quarter) (Ab4 quarter) (G##4 eighth) (rest sixteenth)
NEW_MEASURE
NEW_MEASURE
([Bbb5, Db5, C5] half)
([C#5, E5] half) (C6 minor triad, first inversion, dotted_eighth)
(F#4 half diminished seventh chord, second inversion, eighth)
(D#4 diminished seventh arpeggio, root inversion, quarter)
SET_KEY D minor
SET_KEY C# major
(C harmonic minor descending scale, startNote = Eb4, quarter, length=10)
(Descending Fifths Sequence, G5 minor, quarter, length=15)
(Perfect Authentic Cadence, D#5 major, half)
notes1 chords1 (Ascending Fifths Sequence, G#3 minor, quarter, length=5)
chord (Perfect Authentic Cadence, Eb5 minor, sixteenth) chords1
```

Figure 2: MusAssist Syntax

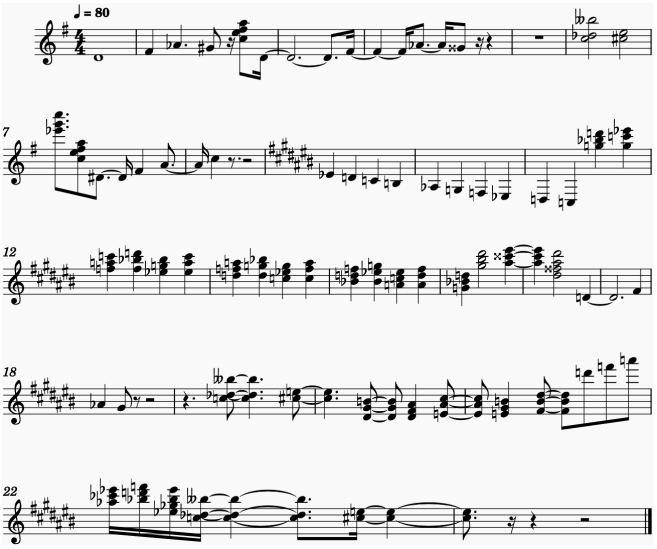

Figure 3: Compiled MusAssist Program in MuseScore

Notice the following in Figure 2:

- The key signature can be changed consecutively arbitrarily many times; only the last will take effect (as seen m. 1 and m. 12).
- Note durations are broken both on the strong beat and on the barline (such as in mm. 2-4).
- Labeled phrases are not notated until the label is referenced, rather than defined.
- The first `NEW_MEASURE` command starts an empty measure, and subsequent commands create empty measures (e.g. mm. 4-5).

5. COMPILER STRUCTURE

The MusAssist compiler is written in Haskell. Its high-level structure is as follows:

1. MusAssist concrete syntax is parsed into abstract syntax, represented as Haskell algebraic data types (ADTs). Parser combinators were chosen for their flexibility and easy customization. Parsec, an industrial strength parser library, is used, and Parsec's helper module Token handles lexing. The parse preserves the abstraction level of all templates.

2. All templates, now represented as ADTs, are expanded until the granularity reaches the note level. The result of the intermediate stage is abstract syntax whose abstraction level matches that of the target language MusicXML.

3. The low-level abstract syntax resulting from the fully expanded templates is translated to MusicXML. This step contains the temporal logic that subdivides notes and rests across barlines and strong beats.

The resulting MusicXML file can then be opened in music notation software for viewing and further editing.

6. TEMPLATE EXPANSION LOGIC

MusAssist's most powerful feature is its ability to formalize musical theoretical templates to automate their expansion so the user can specify them at the elevated level of abstraction of the musical theoretical structures they describe. This section summarizes the logic underlying the expansions.

6.1 Generating Notes in a Diatonic Scale

Most MusAssist templates are built upon the diatonic scale. In order to formalize their expansions, we must first implement logic to generate a note in a diatonic scale of specified quality (major or minor), given a positive interval within one octave of the specified tonic. To define the note, we need the note name, octave, and accidental.

To determine note name, we simply transverse up the order of note names (ordered as the C major scale is) from the tonic to the target interval.

The desired octave is either the same as the the tonic's, or one greater than the tonic's if the desired note name comes before the tonic note name in the C major scale. For instance, as seen in Figure 4, the red note names D, E, and F come before G in the C major scale, and the octave number of each is one higher than the tonic in a G major scale.

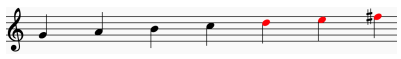

Figure 4: Octave Analysis of G Major Scale

In any key, any perfect interval will have the same accidental as the tonic. To work out the logic behind the accidental of a desired imperfect interval, consider Figure 5. Here, we consider all single-accidental key signature names (even invalid ones that contain double sharps or flats, in order to establish the pattern). Key signature names are grouped under the accidental of the note that is the desired interval from the tonic. For instance, in the key of A♭ , the major second interval from the tonic is B♭. The accidental of B♭ is ♭, so A♭ falls under the ♭ column in Figure 5.

	Major Seconds		
♭	♮	♯	𝄪
F♭	E♭	E	E♯
C♭	B♭	B	B♯
G♭	F	F♯	
D♭	C	C♯	
A♭	G	G♯	
	D	D♯	
	A	A♯	

Figure 5: Accidental of Major Second from Tonic per Key

From Figure 5 we see that given any key, the major second above the tonic has the same accidental as the tonic, except for the any key with E or B in its name. Here, the accidental is "lifted" (i.e. ♭→ ♮, ♮→ ♯, and ♯→ 𝄪).

A similar pattern emerges for minor thirds, major sixths, and minor sevenths. Using the result of this analysis, we can determine the accidentals of the inverse qualities (i.e. major ↔ minor) of the imperfect intervals by either lowering the computed accidental when going from major to minor, or lifting it otherwise.

6.2 Scales

The expansion of all major and natural/harmonic/melodic minor scales is derived from the logic in Section 6.1. The scale is thus generated in relation to the tonic, rather than the specified start note. The tonic is always set below the start note so that the initial interval is positive, with the interval then increasing for ascending scales and descending otherwise until the desired scale length is reached. If the tonic is reached in the scale generation, we reset the tonic to be one octave higher or lower (depending on the scale direction), so that the interval of the next note in the scale is always positive and within one octave of the current tonic. Finally, note that all minor scales are treated as natural; the sixth and/or seventh scale degree is raised appropriately for harmonic and melodic minor after the note is generated.

For the nondiatonic scales (chromatic and whole tone), C is set as the "tonic" for the purposes of determining the octave. Just like diatonic scales, the octave is shifted appropriately if we pass it during the scale generation.

As seen in Figure 7, chromatic scales always "double" the note name, with the directional accidental (sharp for ascending, flat for descending) falling on the second occurrence. There are two exceptions for each direction, marked in red in Figure 7: E and B are "single" notes in the ascending version, as are C and F for the descending version.

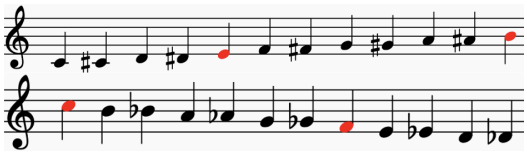

Figure 6: Chromatic Scales

If we are in a single note, we simply move up the scale. Otherwise, we double it and insert an accidental on the second.

Whole tone scales form a similar paradigm. As seen in Figure 7, the ascending whole tone scale is missing the note B, while the descending is missing C.

To determine the next note name, we thus simply traverse up or down the C major scale, excluding the appropriate

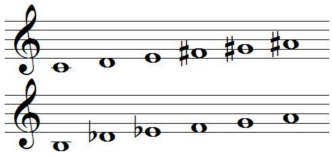

Figure 7: Whole Tone Scales

"skip" note. The accidental changes for both scales are bounded on one end by their respective "skip" notes; the other ends are bounded by E in the ascending scale and F in the descending scale. To determine the next accidental, we thus appropriately lift or lower the current accidental at these boundaries, otherwise leaving it unchanged.

6.3 Chords and Arpeggios

Each note in the chord or arpeggio is generated with the logic from Section 6.1 based on their respective intervals from the tonic (i.e. the chordal root). Chordal thirds, fifths, and sevenths have respective intervals of 2, 4, and 6 from the tonic.

The imperfect chordal intervals are set to major for major and augmented chords, and minor for minor, half diminished, and fully diminished chord. Thus, for augmented chords, the chordal fifth accidental must be lifted, and the chordal seventh accidental must be lowered. For diminished chords, the chordal fifth and seventh accidentals must be lowered, and for half diminished chords, the chordal fifth alone must be lowered.

To handle inversions, recall that the chord (whether triad or seventh) starts out in root position. Let n be the desired inversion value (0 for root, 1 for first inversion, 2 for second, and 3 or third). By incrementing the octaves of the first n tones of the chord in root position, we obtain the correct inversion.

If the chord is an arpeggio, we simply return a series of individual notes, rather than a simultaneous cluster.

6.4 Cadences

A cadence is expanded to the lists of chords it comprises using the logic from Section 6.3.

To do this, a helper function called `generateTriadWithinScale` was created. This function, given the tonic tone and quality (either major or minor), will generate the triad within that diatonic scale given the desired inversion and interval from the tonic (must be between 0 and 6, i.e. within one octave of the tonic) and inversion. Given the tonic quality (i.e. the quality of the scale), `generateTriadWithinScale` first determines the quality of the desired chord. From music theory, it is known that major keys contain the following chords: I-ii-iii-IV-V-vi-vii^o, and minor keys contain the following chords: i-ii^o-III-iv-v-VI-VII. Thus, depending on whether the tonic quality is major or minor, the quality of the triad is computed based on this information.

After this, `generateToneWithinScale` is called to generate the root of the chord. Following this, a chord template for the triad is created, and passed into `expandIntermediateSequence` to get expanded into a `Chord` containing the list of tones and durations. The result of this is the desired triad.

Returning to cadence template expansion, `generateTriadWithinScale` is used to create each triad in the cadence. Notably, none of the cadences in MusAssist utilize seventh chords.

`generateTriadWithinScale` is thus called to generate the chords for the given cadence type based on MusAssist's cadence representations from Figure 1.

A notable edge case that occurred was when creating the V^6_4 chord for deceptive cadences. In order for the cadence to render appropriately, the root of the V^6_4 needs to be an octave number *below* the tonic of the scale. Hence, the special octave cases are `(enumFromTo MusAST.C MusAST.E)` (i.e. C,D, and E, the keys whose fifth scale degree would normally be the *same* octave number as the tonic), and the octave function for these cases is `pred`

Other edge cases included making sure that all V chords were major when calling `generateTriadWithinScale`, no matter the local key quality, since in a cadence, the five chord is always major. Similarly, we want the diminished seventh triad vii^o in the imperfect authentic cadence no matter the local key quality (since we raise the leading tone in minor keys when moving towards the tonic). This means that we also to pass major as the quality parameter to `generateTriadWithinScale` here, in order for this chord to be built off the major seven scale degree.

6.5 Harmonic Sequences

Harmonic sequences were the most complex template to expand. In order to generate a harmonic sequence, three items need to be determined: (1) the interval of the next chord relative to the previous, (2) the inversion of each chord, and (3) the octave number of each chord.

The following diagrams demonstrate the interval pattern for each sequence based on the chord index. All sequences use zero-indexing.

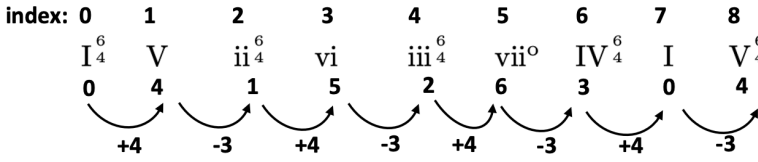

Figure 8: Ascending Fifths Interval Analysis

In each figure, the top row is the chord index, the second row is the chord progression, the third row is the interval of the chord from the tonic, and the bottom row is the interval of each chord from the previous (modulo 7). A clear pattern for both chord inversions and interval changes emerges for each sequence based on the index. Also, recall from Figures **??** – **??** that all descending sequences are written in a descending direction, and ascending sequences are written in an ascending direction. Thus, when descending sequences repeat, their octave number decrements, and when ascending sequences repeat, their octave number increments.

The next step is to determine the octave number of each chord. Recall that the octave number of a chord in the sequence is given by the octave number of the chordal root (no matter the inversion). Also, the octave number of each chord depends only on the note name, not the accidental (since all an accidental does is alter the same note in the staff). By going through each sequence for each of the seven possible notes in a key name (i.e. CDEFFGAB), we

determine the octave number of each chord relative to the octave of the first chord in the sequence. To do this, all chords were converted to root position for the sake of the analysis. For instance, consider Figure 9.

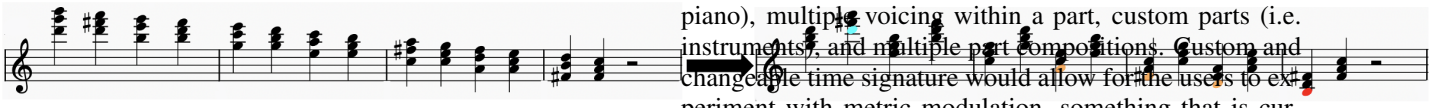

Figure 9: Descending 5-6 Octaves Example

The chordal roots in blue are an octave number above the first chord in the sequence, the chordal roots in yellow are an octave number below, and the chordal root in red is two octave numbers below.

OCTAVE LOGIC

We have thus determined how to find the interval and inversion of a chord in the sequence from the previous chord, and how to find the octave number of a chord in the sequence. Importantly, the interval analysis holds for any representation of these harmonic sequences, as this is what defines the sequence. However, the inversion and octave analysis holds only for MusAssist's chosen representation of the sequences, as a different inversion pattern would deeply alter both of these analyses.

We are now ready to generate the sequences. A generic sequence-generator helper function called `generateSeq n` was written that, given n (the length, or number of chords) in the sequence, generates the sequence. `generateSeq n` determines the next index in the sequence (resetting to zero each time the sequence repeats after 14 chords), either increments or decrements the tonic octave number after each cycle, determines the note name of the next chordal root based on the interval rule from the previous chord for that sequence, and determines the inversion based on the index. The special octave cases and associated octave functions are passed in based on the previously discussed analysis. `generateSeq n` now has enough information to call `generateTriadWithinScale` to generate the desired chord for that index in the sequence, and then recurses to the next chord in the sequence until n goes to zero.

7. CONCLUSION

This paper presents MusAssist, is an external DSL whose Haskell-based compiler translates it to MusicXML that can be loaded into a major music notation software for further editing. Its syntax is simple and models the flow of thought a composer would have when writing music by hand. MusAssist fills a unique niche in the realm of musical DSLs by serving as a music compositional aid that allows the user to write specifications for complex musical templates at the levels of abstraction of the musical structures they describe. MusAssist is not intended to allow to the user to write a fully expressive musical piece, but rather to more easily create musical expressions that would be tedious to write by hand. For this reason, the output MusicXML file of a compiled MusAssist program is intentionally editable, rather than a static PDF format like a language such as LilyPond generates. MuseScore can be further valuable as an education tool to music theory students, allowing them to visualize musical structures from the definitions that they describe in MusAssist.

Ideally, in the future MusAssist would support more complex musical states and elements including custom time signature and mid-composition time signature changes (similar to the behavior currently implemented for key signatures), clef changes within a part, multiple-clef parts (e.g. piano), multiple voicing within a part, custom parts (i.e. instruments), and multiple part compositions. Custom and changeable time signature would allow for the users to experiment with metric modulation, something that is currently impossible with the fixed common time setup. Clef changes within a part, both manual and automatic when a note extends too many ledger lines beyond a clef, would allow the score to be more aesthetically formatted and readable for the user. Support for two-clef piano would allow the MusAssist compiler to successfully modify how it generates cadences and harmonic sequences to include the essential baseline, in addition to the harmonization already implemented. Multiple voicing would allow for the user to create more complex musical lines, particularly with counterpoint. The latter two goals (custom parts and multiple parts) are somewhat outside MusAs- sist's goal as a music compositional aid, as this extends beyond the realm of music theory. However, users may enjoy this increased flexibility when composing. Furthermore, in line with its goal of offering the user complex musical templates, it would be ideal for MusAssist to provide support for key modulation; e.g. generating a sequence of chords that successfully modulates from one key to another key. More complex/non-classical chords, such as all flavors of suspended, ninth, eleventh, and thirteen chords (often seen in jazz) are a future goal as well. It would also be ideal for MusAssist to be able to generate scales in any key, of any type (i.e. major, harmonic minor, octatonic etc), and of any length. Finally, MusAssist would in the future support more complex rhythms where any number of notes could be grouped over any number of beats (i.e. triplet, which is three eighths over a quarter note, or a 4:3 rhythm of four eighths over three quarter notes)

8. CITATIONS

All bibliographical references should be listed at the end, inside a section named "REFERENCES".

References must be numbered in order of appearance, *not* alphabetically. Please avoid listing references that do not appear in the text.

Reference numbers in the text should appear within square brackets, such as in [1] or [2, 3].

The reference format is the standard IEEE one. We recommend using BibTeX to create the reference list.

Acknowledgments

At the end of the Conclusions, acknowledgements to people, projects, funding agencies, etc. can be included after the second-level heading "Acknowledgments" (with no numbering).

9. REFERENCES

[1] A. Someone, B. Someone, and C. Someone, "The Title of the Journal Paper," *J. New Music Research*, vol. 12, no. 2, pp. 111–222, 2009.

[2] X. Someone and Y. Someone, *The Title of the Book*. Springer-Verlag, 2004.

[3] A. Someone, B. Someone, and C. Someone, “The Title of the Conference Paper,” in *Proceedings of the 2005 International Computer Music Conference*, Barcelona, 2005, pp. 213–218.