# MODELING BEATS, ACCENTS, BEAMS, AND TIME SIGNATURES HIERARCHICALLY WITH MUSIC21 METER OBJECTS

*Christopher Ariza* *Michael Scott Cuthbert*

Massachusetts Institute of Technology
Music and Theater Arts

## ABSTRACT

The music21 TimeSignature object represents meters hierarchically, through independent display, beam, beat, and accent attributes capable of unlimited partitioning and nesting. This model, designed for applications in computer-aided musicology, accommodates any variety of compound, complex, or additive meters, can report beat position and accent levels, and can algorithmically perform multi-level beaming or various types of metrical analysis. As part of the music21 Python toolkit, the meter module can read input from Humdrum and MusicXML and output to MusicXML and Lilypond.

## 1. INTRODUCTION

Meters and time signatures play diverse and sometimes contradictory roles in notated Western music. A time signature may illustrate not only the duration and beat divisions of a measure, but also dynamic accent, the visual presentation of beams, and even performance style. Accents and beaming can be independent of each other and contradict the time signature. The time signature, in cadenzas or in many modern scores, may reflect a multi-measure group or a duration greater or less than that implied by the notes in the bar. A complete software model of meter must account for this diversity.

Despite over fifty years of modeling Western notated music on the computer, from the earliest work conducted by Lejaren Hiller and Robert Baker [5] to contemporary commercial software systems such as Finale and Sibelius, models of meter are often simply counts of duration units, or beats, per measure. While many systems offer facilities to configure independent display meters and perform manual adjustments far from expected arrangements, such models are often insufficient, not only for automated rendering of notation, but also for computational approaches to examining the metrical relationships of notes in a bar.

The music21 TimeSignature object meets these challenges by offering independent, hierarchical representations of all critical aspects of a meter, including display, beaming, beat, and accent. Each of these four properties are governed by MeterSequence objects, nested structures of MeterTerminals and other MeterSequences that define duration-preserving fractional structures. This model accommodates any variety of compound, complex, or additive meters, can report beat position and accent levels, and can algorithmically perform multi-level beaming and various types of metrical analysis. While such knowledge may be implicit in some scores, this data is commonly removed or ignored in software data models. The TimeSignature object provides easy access to this data, and can further transform the presentation of events contained within its bar. No similar design, accommodating all of these features, has been documented.

This more complete model of meter has broad applications. While problems such as signal-based beat tracking or meter induction are outside of the domain of this model, the results of such processes can be better encoded, analyzed, and visualized with this framework.

This design, part of the larger music21 toolkit for computer-aided musicology, is implemented in Python. The music21 toolkit is a new, object-oriented approach to modeling and processing symbolic music representations. As part of the music21 toolkit, the meter module can read input from Humdrum/Kern [6], MusicXML [4] or custom-defined formats, write output to MusicXML and Lilypond [9], and take advantage of the numerous high-level objects provided for studying pitch and other musical elements.

## 2. THE PROBLEM OF REPRESENTING METER

Beams often illustrate the hierarchical structure of a bar. While usage is not standardized, beams frequently group beats, but may also show sub-beat groups. Designing objects that can perform multi-level automatic beaming is thus a first step toward an operational model of a TimeSignature. While this is a small subset of the full functionality of TimeSignature, beaming demonstrates the necessity of nested fractional structures that can return meaningful information about the notes in a bar.

In most cases, the numerator and denominator of a time signature establish the number of beats in a measure and their duration, respectively. Common approaches to automatic beaming employ the time signature-specified beat duration to create beaming groups. While this is effective with some rhythms, in other cases such beaming can obscure tuplets or lower-level rhythmic organization.

The case presented below shows that a single hierarchical level insufficiently describes complex beaming. Structures with many hierarchical levels are needed to model time signatures as they are actually used.

Figure 1 provides a measure from the 1910 Durand & Cie. score of Claude Debussy's *Preludes* Book 1, XI, "La Danse De Puck." The division of the bar into two quarter notes is presented in the outermost eighth beam. Additionally, the repeated eighth-note duration groupings are made clear by the break in the sixteenth beam between the 32nd note and the dotted 16th note. This beaming establishes two levels of hierarchical organization: division by two at the highest level, and at the next level division by two again. Each quarter-note division might be seen internally as a bar of 2/8.

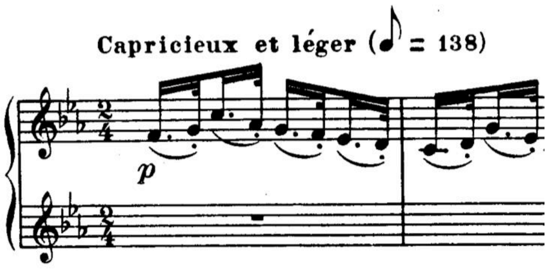

**Figure 1**. Excerpt from Claude Debussy's *Preludes* Book 1, XI

Contemporary notation and music representation systems rarely permit specifying such a hierarchical structure systematically either with beams or with other metrical attributes. While most notation systems permit modifying individual beams freely after their creation, none permit establishing a reusable hierarchical structure of unlimited depth that can be applied to notes contained in a bar.

Conventional beaming algorithms, often limited to one hierarchical level, can not automatically create the beams shown in Figure 1. The output from Finale 2010, illustrated in Figure 2, beams the notes into two groups. The beaming in both Sibelius 6 and MuseScore 0.9 is identical. Lilypond offers great flexibility in manual beaming as well as the ability to specify, within manually-specified beam groups, divisions by durations smaller than the time-signature specified beat. However, as shown in Figure 3, the default beaming divides the phrase into a single level of four beam groups. Lastly, while the FOMUS system [11] performs multi-level beaming, configuration is not integrated into the time signature model.

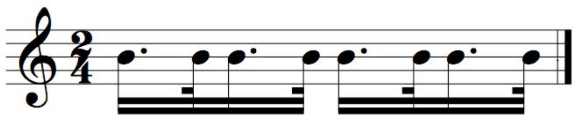
**Figure 2**. Beaming realized in Finale 2010

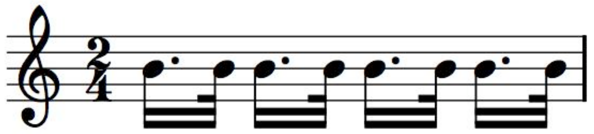
**Figure 3**. Beaming realized in Lilypond 2.12

Beyond simply specifying a generic 2/4 meter, a `music21` TimeSignature object can specify a MeterSequence for beaming that defines a bar of 2/4 as {1/4+1/4}; each 1/4 constituent, in turn, can be defined as {1/8+1/8}. This structure can be notated as {{1/8+1/8}+{1/8+1/8}}. Applying this TimeSignature to beam the rhythm sequence above, the output, provided from `music21` to MusicXML, is shown in Figure 4. By defining nested hierarchical groups, this model produces beaming that correctly matches the beaming in Figure 1.

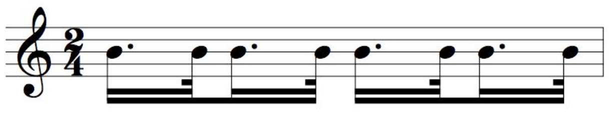
**Figure 4**. TimeSignature beam partitioning by {{1/8+1/8}+{1/8+1/8}}

Just as hierarchical structures can be applied to multi-level beam groups, similar structures can be used to control the display, beat, and accent properties of time signatures.

## 3. OBJECTS FOR ORGANIZING HIERARCHICAL PARTITIONS

Hierarchical models offer useful abstractions of musical structures [1, 2, 7, 8 10, 12]. Hierarchical structures can partition and divide a single duration into one or more parts, where each part is a fraction of the total sum. These parts, in turn, subdivide into one or more constituent parts, a process of partitioning and nesting that can be continued to any depth. Such a structure is defined by the grouping well-formedness rules (GWFR) of Lerdahl and Jackendoff [8 pp. 37-39], specifically, GWFR rules G3, G4, and G5: a group can contain sub-groups, sub-groups cannot be contained by more than one group, and one or more sub-groups must completely fill a group. The fundamental components of TimeSignature, MeterTerminals and MeterSequences, provide flexible representations of such structures.

### 3.1. The MeterTerminal

The MeterTerminal models rhythmic durations. It represents time as a fraction whose numerators may be any positive integer greater than zero and whose denominators may be any $n$ where $n = 2^x$, for $x$ between 1 and 7 (the minimum of 1/128th being a practical, not a structural, lower limit). The fraction specified is applied to a whole note (independent of tempo). The fraction 1/4 thus equals one quarter note, or 1.0 quarter lengths (QLs). The fraction 1/16 is equal to one 16th note, or 0.25 QLs. The MeterTerminal duration is stored as a `music21` Duration object. Each MeterTerminal additionally stores a weight, a numerical value that can be interpreted as an accent value or in a variety of other contexts.

Figure 5 demonstrates basic functionality of the MeterTerminal via a Python interactive session. The `subdivide()` method partitions a MeterTerminal into one or more MeterTerminals that are contained in a MeterSequence, defined below. The arguments given to `subdivide()` may be a single integer specifying the number of equal partitions, a list of desired numerators (where a best-fit denominator is found if possible), or a list of MeterTerminal fractions specified as strings. If the sum of the requested partitions is not equal to the MeterTerminal, `subdivide()` raises an error. Since `subdivide()` returns a MeterSequence, re-assigning the output to a MeterTerminal is a common usage.

```
>>> from music21 import meter
>>> mt = meter.MeterTerminal('3/4')
>>> mt.numerator, mt.denominator
(3, 4)
>>> mt.duration
<music21.duration.Duration 3.0>
>>> mt.duration.quarterLength
3.0
>>> mt.subdivide(3)
<MeterSequence {1/4+1/4+1/4}>
>>> mt.subdivide([3,3])
<MeterSequence {3/8+3/8}>
>>> mt.subdivide(['1/4','4/8'])
<MeterSequence {1/4+4/8}>
```

**Figure 5**. Usages of a MeterTerminal Object

## 3.2. The MeterSequence

The MeterSequence is a subclass of MeterTerminal whose numerator and denominator values are determined solely by the sum of an ordered list containing one or more MeterTerminals and/or other MeterSequences. MeterSequences have durations, though these durations, like their numerators and denominators, are immutable after the object has been created. The MeterSequence, like a MeterTerminal, thus has a fixed duration specified as a fraction, but unlike a MeterTerminal it can be partitioned to any hierarchical depth.

Through operator overloading, the top-level partitions of a MeterSequence can be accessed like Python lists. Like a list, the MeterSequence has a length, accessed by the Python built-in function `len()`; this value equals the number of top-level partitions. Also similar to a list, top-level partitions can be iterated and accessed directly with indexes starting from zero.

The `weight` of a MeterSequence is always the sum of the weights of its constituent MeterTerminals or MeterSequences. When directly setting the `weight` of a MeterSequence, fractional weights are assigned to stored partitions proportional to the fraction of the total duration each partition occupies.

MeterSequences, unlike MeterTerminals, can be re-partitioned in-place by calling the `partition()` method. Using arguments in the same forms as used with `subdivide()`, the `partition()` method replaces existing stored MeterTerminals and MeterSequences with new partitions. The total duration of new partitions must remain the same as before. Repartitioning thus does not change the MeterSequence's numerator or denominator. As a subclass of MeterTerminal, MeterSequence also has a `subdivide()` method to return a newly partitioned MeterSequence.

Additional properties and methods provide access to nested data within the MeterSequence. The `flat` property returns a new MeterSequence built from the lowest-defined MeterTerminals. Similarly, the `flatWeight` property returns a list of lowest-defined MeterTerminal weights. The `getLevel()`, `getLevelSpan()`, and `getLevelWeight()` methods provide slices through hierarchical levels, returning partitions, partition time spans, or weight values, respectively, for a given depth value starting at zero. The usage of these methods is less complex than it might seem, as Figure 6 demonstrates.

Finally, a MeterSequence has three methods for accessing the index value, temporal boundaries, and contained depth of a partition based on a supplied QL. The `positionToIndex()` method returns the index of the topmost partition. The `positionToSpan()` method returns the start and end values of the partition within which the provided QL falls. Lastly, the `positionToDepth()` method returns the number of partitions that start at or below the supplied QL.

Figure 6 shows a MeterSequence partitioning and subdividing a 3/4 meter.

```
>>> from music21 import meter
>>> ms = meter.MeterSequence('3/4')
>>> ms.partition([3,3])
>>> ms
<MeterSequence {3/8+3/8}>
>>> len(ms)
2
>>> ms[0]
<MeterTerminal 3/8>
>>> ms[0] = ms[0].subdivide([3,3])
>>> ms
<MeterSequence {{3/16+3/16}+3/8}>
>>> ms[1] = ms[1].subdivide([1,1,1])
>>> ms
<MeterSequence {{3/16+3/16}+{1/8+1/8+1/8}}>
>>> ms[1][0]
<MeterTerminal 1/8>
>>> ms.depth, ms[0].depth
(2, 1)
>>> ms.flat
<MeterSequence {3/16+3/16+1/8+1/8+1/8}>
>>> ms.flatWeight
```

```
[0.25, 0.25, 0.16666666666666666,
0.16666666666666666, 0.16666666666666666]
>>> ms.getLevel(0)
<MeterSequence {3/8+3/8}>
>>> ms.getLevel(1)
<MeterSequence {3/16+3/16+1/8+1/8+1/8}>
>>> ms.positionToSpan(.5)
(0, 1.5)
>>> ms.getLevel(1).positionToSpan(.5)
(0, 0.75)
```

**Figure 6**. Usages of the MeterSequence Object

## 4. THE TIMESIGNATURE OBJECT

As Lerdahl and Jackendoff note, "metrical structure as such does not possess any inherent grouping" [7 p. 123]. Supporting this claim, the model presented here permits diverse groupings within a meter, isolated among independent display, beaming, beat, and accent attributes. We limit the structures here to those contained within a single bar; this limitation does not rule out the application of extended models to larger durations.

The TimeSignature object contains four MeterSequences, stored as attributes: `display`, `beat`, `accent`, and `beam`. Each MeterSequence shares the same initial duration relationship but can be configured independently, providing a wide diversity of arrangements. The applications of each MeterSequence, as well as high-level TimeSignature methods, are examined below. As with the MeterSequence, the TimeSigature has a fixed numerator and denominator, as well as a fixed duration, accessed from `totalLength` (returned as a number expressed in QLs) and `barDuration` (returned as a Duration object). As a high-level interface, TimeSignature provides sensible defaults for all common (and many uncommon) meters, as well as output in partial or complete notated representations through the `lily` (for Lilypond) and `mx` (for MusicXML) properties.

Figure 7 demonstrates one way of creating a 5/8 TimeSignature. A string representation of a meter is required as an initial argument for creating the object. Additive string representations, such as 3+2/8 or 2/16+3/8, are permitted. If provided, these implied partitions are applied to the `display` MeterSequence. An optional argument can override defaults and set initial partitioning, applied to `beam`, `accent`, and `beat` MeterSequences. The `load()` method can be called to reinitialize the TimeSignature and all contained MeterSequences.

```
>>> from music21 import meter
>>> ts = meter.TimeSignature('5/8', \
...                          ['2/8', '3/8'])
>>> ts.numerator, ts.denominator
(5, 8)
>>> ts.barDuration
<music21.duration.Duration 2.5>
>>> ts.display
<MeterSequence {5/8}>
>>> ts.beam
<MeterSequence {2/8+3/8}>
>>> ts.accent
<MeterSequence {2/8+3/8}>
>>> ts.beat
<MeterSequence {2/8+3/8}>
```

**Figure 7**. Usages of the TimeSignature Object

### 4.1. The `display` Attribute

Many notation systems permit assignment of a different meter for display than for beat or beam grouping. This approach is accommodated in the TimeSignature object with the `display` attribute, a MeterSequence dedicated to the displayed time signature value. This attribute interprets the highest-level partitions as the displayed meter, where partitions are presented as individual meters connected by addition. Lower-level MeterSequence partitions and weights are not used for this attribute.

In Figure 8 two different TimeSignature objects are created with an initial meter of 5/8. The `display` MeterSequences are independently configured. The first (`ts1`) creates three partitions, displayed as summed meters. The second (`ts2`) sets the `summedNumerator` attribute of the `display` MeterSequence to True: this setting presents numerators summed over a common denominator. These TimeSignatures are applied to two `music21` Measures, each filled with Note objects with durations of 0.5 QLs. These Measures are appended to a `music21` Stream, an offset positioned, list-like container capable of generating complete notation in MusicXML and Lilypond via its `show()` method. The MusicXML output is presented in Figure 9.

```
ts1 = meter.TimeSignature('5/8')
ts1.display.partition(['3/16','1/8','5/16'])

ts2 = meter.TimeSignature('5/8')
ts2.display.partition(['2/8', '3/8'])
ts2.summedNumerator = True

s = stream.Stream()
for ts in [ts1, ts2]:
    m = stream.Measure()
    m.timeSignature = ts
    n = note.Note('b')
    n.quarterLength = 0.5
    m.repeatAppend(n, 5)
    s.append(m)
s.show('musicxml')
```

**Figure 8**. Configuring `display` attributes

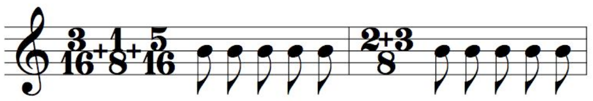

**Figure 9**. The `display` attribute configured as {3/16+1/8+5/16} and {2/8+3/8}

Rather than using the MeterSequence created when initializing the TimeSignature object, the `setDisplay()` TimeSignature method can be used to create a new, independent MeterSequence with no relationship to the duration of the other MeterSequence attributes.

The `display` MeterSequence can be expanded to allow for custom display attributes, such as single number (numerator) presentations, flagged-note denominators, and the medieval and Renaissance meters of C-dot or cut-circle. Although not all such signatures can be fully represented in MusicXML and Lilypond, these representations are useful for computer-aided musicology and will, for example, allow future interchange with the Computerized Mensural Music Editing project [3] and other specialized representations.

### 4.2. The `beam` Attribute

The `beam` attribute provides an independently configurable MeterSequence for TimeSignature beaming. The MeterSequence is interpreted as specifying beam groups, where the top-most partition is the outermost eighth beam break. Subsequent partitions at lower levels specify groups shown with partial beam breaks. MeterSequence weights are not used for this attribute. As demonstrated in Section 2, partial beam breaks illustrate rhythmic groupings and facilitate the reading of rhythms, particularly tuplets mixed with similar durations. Storing reusable beam partitions permits multiple measures sharing a TimeSignature to be automatically beamed to the same partitions.

The TimeSignature `getBeams()` method, called and configured by the Stream `makeBeams()` method, defines the automatic beaming algorithm. This method uses the TimeSignature's `beam` MeterSequence to apply beams to `music21` Note objects in a bar. A beamed Note contains a collective object called Beams consisting of one or more Beam objects, each describing the state of a beam segment. On output generation, these objects render beams in MusicXML or Lilypond.

Figure 10 provides Python code to configure four different meters at various beaming levels. The first (`ts1`) has one partition subdivided into four beam groups. The second (`ts2`) has three equally sized partitions. The third (`ts3`) has three partitions each subdivided into two groups. The fourth (`ts4`) demonstrates the use of three hierarchical beaming levels. Figure 11 shows the differences in the output of these four representations applied to a bar of 32nd notes in 3/4.

```
ts1 = meter.TimeSignature('3/4')
ts1.beam.partition(1)
ts1.beam[0] = ts1.beam[0].subdivide(['3/8',
             '5/32', '4/32', '3/32'])

ts2 = meter.TimeSignature('3/4')
ts2.beam.partition(3)

ts3 = meter.TimeSignature('3/4')
ts3.beam.partition(3)
for i in range(len(ts3.beam)):
    ts3.beam[i] = ts3.beam[i].subdivide(2)

ts4 = meter.TimeSignature('3/4')
ts4.beam.partition(['3/8', '3/8'])
for i in range(len(ts4.beam)):
    ts4.beam[i] = ts4.beam[i].subdivide(
        ['6/32', '6/32'])
    for j in range(len(ts4.beam[i])):
        ts4.beam[i][j] = \
        ts4.beam[i][j].subdivide(2)

s = stream.Stream()
for ts in [ts1, ts2, ts3, ts4]:
    m = stream.Measure()
    m.timeSignature = ts
    n = note.Note('b')
    n.quarterLength = 0.125
    m.repeatAppend(n, 24)
    s.append(m.makeBeams())
s.show('musicxml')
```

**Figure 10**. Configurations of the `beam` attribute

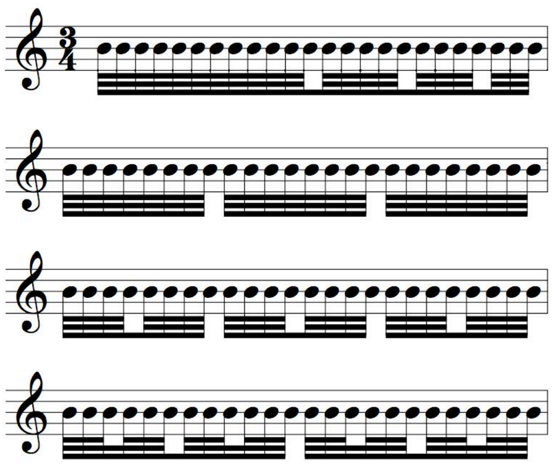

**Figure 11**. The `beam` attribute configured as `ts1` {3/8+5/32+4/32+3/32}, `ts2` {1/4+1/4+1/4}, `ts3` {{1/8+1/8}+{1/8+1/8}+{1/8+1/8}}, and `ts4` {{{3/32+3/32}+{3/32+3/32}}+{{3/32+3/32}+{3/32+3/32}}}

Figure 12 provides Python code to configure two TimeSigature `beam` attributes to obtain the same beaming used in two measures of the solo violin part from the 1936 Universal Edition score (UE 12195) of Alban Berg's *Violinkonzert*. The MusicXML output is presented in Figure 13. The Universal Edition score divides the lower-level beam groups differently in each bar, differences that are modeled by `music21`'s TimeSignature.

```
ts1 = meter.TimeSignature('6/8')
ts1.beam.partition(2)

ts2 = meter.TimeSignature('6/8')
ts2.beam.partition(2)
```

```
for i in range(len(ts2.beam)):
    ts2.beam[i] = ts2.beam[i].subdivide(3)
```

**Figure 12**. TimeSignature configuration for the violin part from Alban Berg's *Vioinkonzert*, measures 135-136

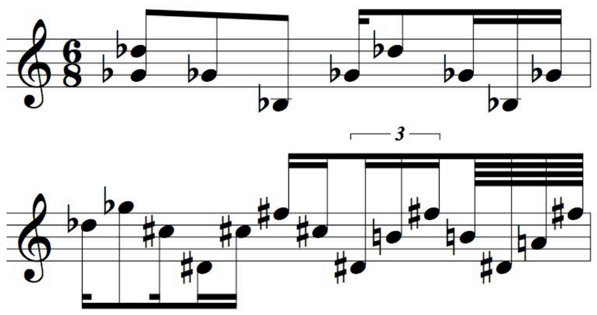

**Figure 13**. The `beam` attribute configured as `ts1` {3/8+3/8} and `ts2` {{1/8+1/8+1/8}+{1/8+1/8+1/8}} for the violin part from Alban Berg's *Vioinkonzert*, measures 135-136

## 4.3. The `beat` Attribute

The `beat` attribute provides an independent MeterSequence to partition the bar into beat divisions. Although Lerdahl and Jackendoff assert that "beats must be equally spaced" and that "fundamental to the idea of meter is the notion of periodic alternation of strong and weak beats" [8 p. 19], the use of a `beat` MeterSequence here is more general and is independent of metrical accent. While beat is usually linked to metrical accent, independent MeterSequences permit greater representational flexibility.

Applications of the `beat` attribute include filtering notes or and labeling them by their beats. Lower-level partitions have applications for analysis, as shown below. MeterSequence weights, not explored here, are potentially useful for related analytical tasks. The `getBeat()` method of TimeSignature provides the beat count (starting from 1) for a note (or any other musical event) given a QL into a bar. For instance, for a 3/8 bar with three equal beats, `getBeat(0.5)` returns 2. The `getBeatProgress()` TimeSignature method returns the beat count as well as the QL difference from the start of that beat, so in the previous example, `getBeatProgress(0.75)` returns (2, 0.25). Figure 14 uses these methods on a chorale by J. S. Bach, BWV 366, (from a corpus of works distributed with `music21`) to find and label all the raised seventh scale degrees in this d-minor piece. Found notes are labeled with voice part, measure number, and beat. The resulting MusicXML output, shown in Figure 15, illustrates that in this case, Bach uses raised seventh scale degrees only on beats one and three, the two strongest beats.

```
import music21
from music21 import corpus, meter, stream

score = corpus.parseWork('bach/bwv366.xml')
ts = score.flat.getElementsByClass(
    meter.TimeSignature)[0]
ts.beat.partition(3)

found = stream.Stream()
offsetQL = 0
for part in score:
    found.insert(offsetQL,
        part.flat.getElementsByClass(
        music21.clef.Clef)[0])
    for i in range(len(part.measures)):
        m = part.measures[i]
        for n in m.notes:
            if n.name == 'C#':
                n.addLyric('%s, m. %s' %
                    (part.id[0],
                    m.measureNumber))
                n.addLyric('beat %s' %
                    ts.getBeat(n.offset))
                found.insert(offsetQL, n)
                offsetQL += 4

found.show('musicxml')
```

**Figure 14**. Using `getBeat()` to label found notes

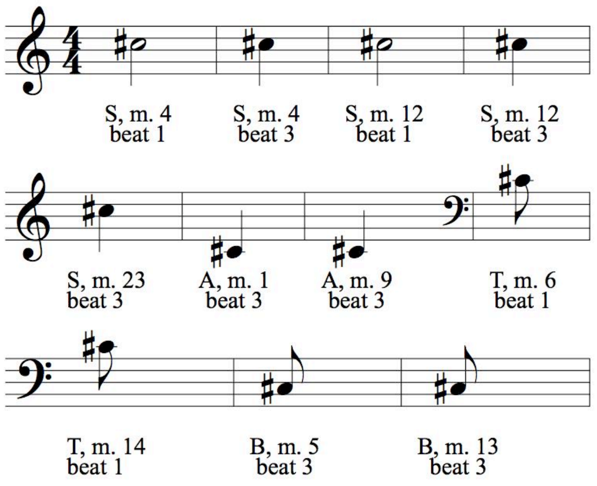

**Figure 15**. Collected results showing part, measure, and beat

Beats can be also represented as composed from lower-level hierarchical units. Lerdahl and Jackendoff's *metrical analysis* [7 p. 120] employs multiple levels of hierarchical beat structures to partition a bar into "dot levels." An example of their analysis is presented in Figure 16.

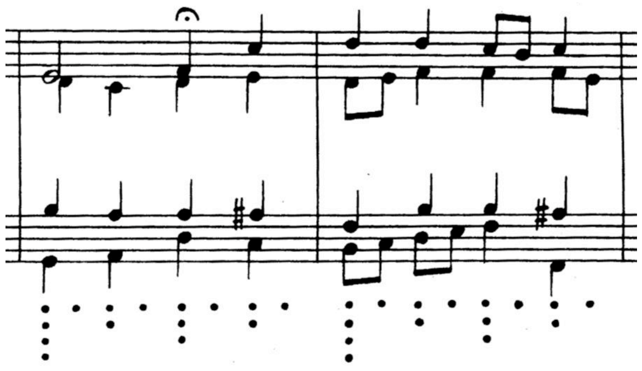

**Figure 16**. Metrical analysis excerpt from Lerdahl and Jackendoff [7 p. 121]

In the more general context of a MeterSequence, metrical analysis dot levels can be obtained by examining the depth, or the number of hierarchical levels active at a QL value. A QL, quantized to the start of the lowest-level partition within which the QL falls, is used to find the number of partitions at all levels that share this start QL. The TimeSignature method `getBeatDepth()` provides these values. The Python code in Figure 17 annotates notes from the bass part of a chorale by J. S. Bach, BWV 281, with the number of dots specified by a 4/4 time signature subdivided in halves down to the eighth note level. The MusicXML output is presented in Figure 18.

```
import music21
from music21 import corpus, meter

score = corpus.parseWork('bach/bwv281.xml')
partBass = score.getElementById('Bass')
ts = partBass.flat.getElementsByClass(
     meter.TimeSignature)[0]

ts.beat.partition(1)
for h in range(len(ts.beat)):
    ts.beat[h] = ts.beat[h].subdivide(2)
    for i in range(len(ts.beat[h])):
        ts.beat[h][i] = \
            ts.beat[h][i].subdivide(2)
        for j in range(len(ts.beat[h][i])):
            ts.beat[h][i][j] = \
                ts.beat[h][i][j].subdivide(2)

for m in partBass.measures:
    for n in m.notes:
        for i in range(
            ts.getBeatDepth(n.offset)):
            n.addLyric('*')

partBass.measures[0:7].show('musicxml')
```

**Figure 17**. Metrical analysis applied to notes based on `beat` attribute depth

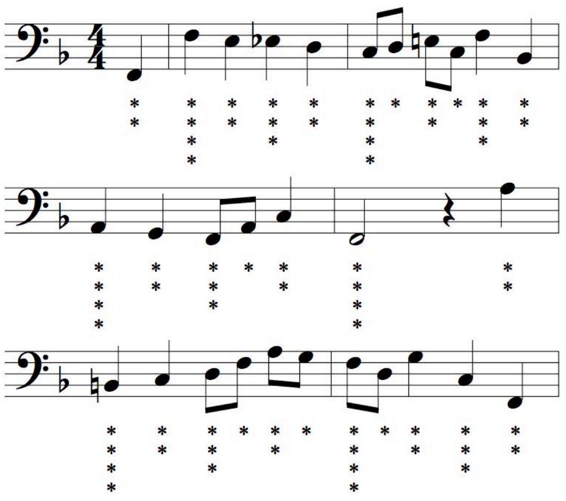

**Figure 18**. Bass part annotated with metrical analysis

## 4.4. The `accent` Attribute

The `accent` attribute provides an independent MeterSequence to partition the bar into accent groups, where each top-level partition is given a weight proportional to its metrical stress. Weights can be set directly or through the `setAccentWeight()` TimeSignature method. Hierarchical weights, while not demonstrated here, may offer valuable representational opportunities. The `accent` attribute may be used for modelling a dynamic stress, as demonstrated here, or for other interpretations of the concept of accent.

The `getAccentWeight()` TimeSignature method provides, for a supplied QL into a bar, the weight active at a specified hierarchical level. This method is demonstrated in Figure 19, where the bass part of the chorale used in Figure 14 is annotated with articulation marks to reflect accent levels. Rather than the displayed meter of 3/4, here the beat and accent attributes are (incorrectly) partitioned into 6/8, or groupings of 3/8 + 3/8. The `getBeatProgress()` method of TimeSignature is used to select only the notes that begin at the start of beat divisions. The MusicXML output is presented in Figure 20.

```
from music21 import corpus, meter, articulations

score = corpus.parseWork('bach/bwv366.xml')
partBass = score.getElementById('Bass')

ts = partBass.flat.getElementsByClass(
    meter.TimeSignature)[0]
ts.beat.partition(['3/8', '3/8'])
ts.accent.partition(['3/8', '3/8'])
ts.setAccentWeight([1, .5])

for m in partBass.measures:
    lastBeat = None
    for n in m.notes:
        beat, progress = ts.getBeatProgress(
            n.offset)
        if beat != lastBeat and progress == 0:
            if n.tie != None \
                and n.tie.type == 'stop':
                continue
            if ts.getAccentWeight(n.offset) == 1:
                mark = \
                    articulations.StrongAccent()
            elif ts.getAccentWeight(n.offset) \
                == .5:
                mark = articulations.Accent()
            n.articulations.append(mark)
            lastBeat = beat
        m = m.sorted

partBass.measures[0:8].show('musicxml')
```

**Figure 19**. Using `getBeatProgress()` and `getAccentWeight()` to add articulation marks

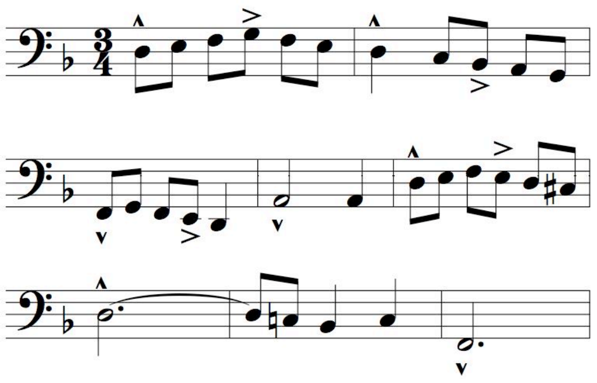

**Figure 20**. Bass part with marks added to articulate {3/8+3/8}

## 5. FUTURE WORK

The TimeSignature model provides a new and powerful tool for analyzing and configuring notes in a bar. In the context of the `music21` toolkit, the TimeSignature object can be used to determine the beat and implied accent of an event in a bar and then extract and process this data. This information is significant in large-scale computational musicology tasks. Similarly, the TimeSignature can be used for automatic beaming algorithms and the creation, performance, and notation of meter-implied accents. Features of this model can be implemented in higher-level systems and interfaces, and would provide significant improvements to commercial notation packages and digital audio workstations.

Future development of this model will support meta- and hyper-meters, thereby providing multi-measure groupings and automatic beaming across bar-lines. Support for independent beaming and accent for multiple-voices within a bar will also be developed.

The TimeSignature model will be used to solve problems in musicology by studying norms of behavior of a large musical corpus. Such research, planned as part of the `music21` project, will include documenting relationships between accent and harmonic structure in Renaissance and common-practice music, and the role of notational variants in identifying scribes and composers.

## 6. ACKNOWLEDGMENTS

Development of TimeSignature and the `music21` toolkit is conducted as part of a multi-year research project funded by the Seaver Institute.

## 7. REFERENCES

[1] Balaban, M. 1992. "Music Structures: Interleaving the Temporal and Hierarchical Aspects in Music." In *Understanding Music with AI: Perspectives on Music Cognition*. M. Balaban, K. Ebcioglu and O. E. Laske, eds. Cambridge: AAAI Press / MIT Press. 31-48.

[2] Buxton, W. and W. Reeves, R. Baecker, L. Mezei. 1978. "The Use of Hierarchy and Instance in a Data Structure for Computer Music." *Computer Music Journal* 2(4): 10-20.

[3] Dumitrescu, T. 2001. "Corpus Mensurabilis Musice 'Electronicum': Toward a Flexible Electronic Representation of Music in Mensural Notation." *Computing in Musicology* 12: 3-18.

[4] Good, M. 2001. "An Internet-Friendly Format for Sheet Music." In *Proceedings of XML 2001*.

[5] Hiller, L. A. and R. A. Baker. 1965. "Automated Music Printing." *Journal of Music Theory* 9(1): 129-152.

[6] Huron, D. 1997. "Humdrum and Kern: Selective Feature Encoding." In *Beyond MIDI: the Handbook of Musical Codes*. E. Selfridge-Field, ed. Cambrdige: MIT Press. 375-401.

[7] Lerdahl, F. and R. Jackendoff. 1977. "Toward a Formal Theory of Tonal Music." *Journal of Music Theory* 21(1): 111-172.

[8] Lerdahl, F. and R. Jackendoff. 1983. *A Generative Theory of Tonal Music*. Cambridge: MIT Press.

[9] Nienhuys, H. and J. Nieuwenhuizen. 2003. "LilyPond, a system for automated music engraving." *Proceedings of the XIV Colloquium on Musical Informatics (XIV CIM 2003)*.

[10] Polansky, L. and P. Burk, D. Rosenboom. 1990. "HMSL (Hierarchical Music Specification Language): A Theoretical Overview." *Perspectives of New Music* 28(1-2): 136-178.

[11] Psenicka, D. 2007. "FOMUS, a Music Notation Software Package for Computer Music Composers." In *Proceedings of the International Computer Music Conference*. San Francisco: International Computer Music Association. 75-78.

[12] Smaill, A. and G. Wiggins, M. Harris. 1993. "Hierarchical Music Representation for Analysis and Composition." *Computers and the Humanities* 27(1): 7-17.