THE MUSIC21 STREAM: A NEW OBJECT MODEL FOR REPRESENTING, FILTERING, AND TRANSFORMING SYMBOLIC MUSICAL STRUCTURES

Christopher Ariza *Michael Scott Cuthbert*

Music and Theater Arts Section
Massachusetts Institute of Technology

ABSTRACT

The Stream is a new object model of symbolic musical structures deployed as part of the `music21` toolkit for computational musicology. Streams are containers that allow for multiple hierarchical and flat representations of a musical work. This flexibility avoids the pitfalls of existing tree-based and event-list-based models and allows for easy searching, discovery of the context of an object (such as a note's current clef), and transformation (such as transposition or duration stretching). This paper describes the `music21` Stream, and offers several examples to show how this multivalent representation is necessary for researchers working on problems in computer-aided musical analysis.

1. INTRODUCTION

Symbolic musical data is messy. While a note is easily translated into a single parameterized event, a staccato note in a measure in a part, tied from a previous measure and in the middle of a slur (that is, conceived as represented in a score) is a much more difficult entity to represent as a data structure. Scores and written notation hold important information that is too easily lost in symbolic music representations. Symbolic representations challenge software designers to balance retaining specification depth with ease of navigation and processing.

All systems for generating and processing musical structures rely on some fundamental representation of musical events. Some storage units that appear frequently are event lists and various hierarchical structures such as trees. While some simple structures (such as the event-list-based scores and files of Csound and MIDI) need only associate event parameters with an instrument, more complex score and notation-based structures that encode symbolic music necessitate relating temporally and/or hierarchically distant events. More than just a data specification, an object model can store and generalize these relationships at a high-level with intuitive functionality.

The `music21` Stream and its related objects form a new model for representing and manipulating collections of complex musical structures. This model (1) stores, orders, and records the position and duration of elements, (2) lets the same element appear in different places in multiple containers (such as in an instrumental part and a full score), (3) can switch between hierarchical and flat representations, (4) lets users iterate over and isolate classes of elements (e.g., clefs, Roman numerals, or rests) that are of particular interest, and (5) lets searches be aware of their contexts. While other researchers have addressed conceptual problems in the design of systems for symbolic music processing beyond applications in notation software, none prior to `music21` have put a comprehensive object model into practice.

The `music21` system offers this object model within an open-source, Python framework. This toolkit is designed for computer-aided musicology and the creation and manipulation of symbolic music data [4]. The system can be used for computational musicology on large repertories, music information retrieval, data mining and machine learning, notation editing and manipulation, and algorithmic composition. The Stream model, the focus of this paper, is the fundamental container of all elements in `music21`. Streams contain numerous other object models for musical elements (e.g., Notes, Chords, Clefs) and musical analysis (e.g., RomanNumerals). The `music21` system, in addition to defining these objects, reads a wide variety of symbolic encodings (e.g., MusicXML, ABC, Musedata, Humdrum, MIDI), outputs music notation in MusicXML or Lilypond, and creates MIDI files. Although the mechanics of data import and export are not the focus of this presentation, that the Stream can represent data brought in from such diverse formats shows its utility and flexibility. More details about the Stream, including extensive documentation, are available at mit.edu/music21.

2. HISTORICAL PERSPECTIVES

The importance of hierarchical structures in modeling musical events and notation elements has been frequently suggested. Lejaren Hiller, for example, claims that "hierarchical structure is the fundamental architectural principle that makes a musical work into a coherent whole" [7]. Singular dependency on hierarchical representations, however, can create flawed designs. As Eleanor Selfridge-Field states, "while it is often convenient for analytical purposes to view music as hierarchical in nature, it is only within the bounds of certain specific attributes that musical information can be thought of as being hierarchical" [14]. For instance, slurs and cross-staff beams break most hierarchies, and data about the pitch class distribution of a piece is extracted without regards to any hierarchy. Furthermore, the fundamental hierarchy of a piece may be different to the analyst than to, say, the engraver needing to lay out the work on a page. Thus, both flat and hierarchical representations, as well as representations that cut across the two, are necessary to model the complexity of actual scores.

In one of the first applications of object-oriented design for musical event structures, Buxton [2] proposes a model where event lists, as events themselves, can be deployed in multiple instances at multiple hierarchical levels. Buxton describes a design, where "each appearance of a particular sub-score constitutes an instance rather than a master copy of that sub-score," and further, "any changes to the original are reflected in each instance" [3]. Combining "arbitrary hierarchical (tree) structures" with object-oriented deployment of event list instances adds flexibility to the model. As Buxton notes, this design gives "the ability to deal with any (user-defined) sub-set of a score in the same manner, and with the same ease, as with a single note" [3]. This functionality, while just as necessary over thirty years after Buxton wrote, is still not readily available.

Other researchers have implemented similar models to solve the same problem. Pope [12] describes an event-list model closely related to that proposed by Buxton and colleagues [4]. In Pope's model, "event lists are events themselves and can therefore be nested into trees… they can also map their properties onto their component events" and "an event can be 'shared' by being in more than one event list at different relative start times and with different properties mapped onto it" [12]. A similar arrangement is found in Daniel Oppenheim's Dmix system, where EventLists are modeled as collections of TerminalEvents (single events or messages) or embedded EventLists [11]. The `music21` Stream offers closely related functionality, but with a much broader range of applications.

More recently, numerous musical data storage formats that define hierarchical event and notation structures have been deployed. MusicXML [5], Musedata [6], Humdrum/Kern [8], and ABC [11] are examples. All of these, however, are designed for storage and interchange; none offer a practical object-model with integrated functionality for manipulating and querying data. All offer various ways of defining notes, measures, voices, and parts, yet provide no way to navigate, filter, or transform these data structures, let alone to parse them. The `music21` object model offers both a representation and a host of integrated functionality, and does so in a convenient, powerful, and widely-used programming language.

3. DESIGN FEATURES

The Stream model has already demonstrated its utility in a variety of research tasks; an example of its power closes this paper. The focus of this presentation, however, is on the design, composition, and interaction of the objects. This paper uses Python examples to demonstrate critical functionality; `assert` statements highlight important values. As code readability is an important design feature, somewhat extended code excerpts are provided. To execute the code, import `music21` with the Python statement, "`from music21 import *`". All notated output, when provided, is created via `music21` MusicXML output that has been imported into Finale.

3.1. Storing, Ordering, and Timing Elements

The most basic structure for musical events is one that permits elements (events, parameters, notations) to reside in containers. Generally, multiple parallel containers are deployed to create musical parts. Often, containers contain sub-containers, such as when measures reside in parts, or when voices reside in measures.

The `music21` Stream is the common container class from which all specialized containers are derived. The Stream itself is a subclass of the Music21Object, which provides its subclasses with functionality such as awareness of its location(s) and the ability to be placed within a Stream. Python objects that are not Music21Objects, such as database or web interfaces, may also be contained in Streams, though they must first be placed in an ElementWrapper, a Music21Object subclass. Since Streams are Music21Objects, they too can be contained within other Streams. Commonly used Stream subclasses include Score, Part, and Voice objects. Figure 1 diagrams the class inheritance of common `music21` objects, and groups them into categories of containers and elements.

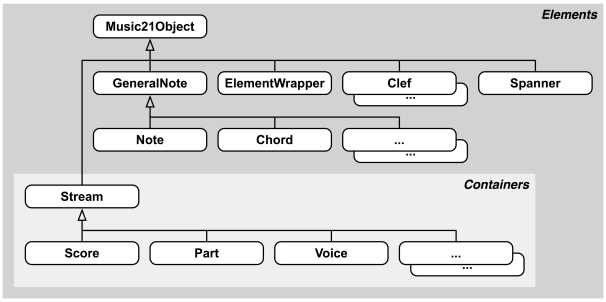

Figure 1. Object inheritance for representative `music21` elements and containers (though containers are also elements).

Streams are in many ways like Python lists, and store references to stored objects in a Python list exposed to the user as `elements`. Elements in Streams, however, are not only ordered, but also have a temporal position and a duration. This permits the order of components in a Stream to be independent of their conceptual or temporal position in the score Stream (though by default Streams are automatically sorted to reduce this potential complexity). Unlike a list, Streams handle elements positioned in overlapping and/or simultaneous arrangements. Ordering is handled with Python list-like indices and associated methods. Timing is measured with offsets, floating point values representing distance from the start of the container. Offsets are generally given in symbolic quarter length values (e.g., 1.0 is one quarter note, 2.0 is one half note, 0.25 is one sixteenth note) but could instead represent time in seconds from a start time or distance in inches or millimeters from a margin.

The Music21Object is subclassed for all elements found in Streams. Nearly all notational entities, both sounding and non-sounding (e.g., Clef, TimeSignature, Note, Chord, Mordent, Metadata) are Music21Objects. A design feature of `music21` is that all sounding and notation elements, from bar lines to instruments, define a duration (which can be zero) and are positioned in Streams at specific offsets. For instance, a Metadata object, specifying a composer's name, might be positioned at offset 40.0 with a duration of 60.0, indicating that only that span was authored by a particular composer. The Music21Object provides instance attributes, such as `id`, `priority`, and `activeSite`, and is composed of component objects such as DefinedContexts and Duration.

The DefinedContexts object stores weak references to objects that are contextually relevant to a Music21Object. Some of these objects are locations: weak references to a container (i.e., a Stream subclass) with an associated offset. Some of these objects are not locations, defining simple associations. In Python, weak references permit storing the name of an object instance without forcing the object to stay in memory. Thus, if only weak (and no normal) references to an object exist, the object will be garbage collected and the weak reference will return `None`.

Thus, while a Stream contains references to its components stored in the `elements` list, each component (as a Music21Object) stores a collection of locations: a weak reference to a Stream and an offset into that Stream. A Stream can thus sort each component by obtaining the offset relevant to itself. Sorting of simultaneous events uses subclass-specified sort order values. For instance, by default Clefs sort before TimeSignatures, which in turn sort before Notes.

The Duration object defines a time span over which a Music21Object is relevant or active. In many cases, this is the symbolic representation of a notation element (e.g., a sixteenth note). In other cases, this is the accumulated time of a larger collection of events (e.g., the span of an entire Stream such as a Measure).

Figure 2 summarizes the types of object compositions and references used in the arrangement of a Note in a Stream. Dotted lines are weak references, solid lines are normal references. Closed diamonds are object compositions and suggest lifetime responsibility for the component. Open diamonds are object aggregations and do not suggest lifetime responsibility. A Note, for example, is composed of a Pitch, Duration, and DefinedContexts objects (as well as others). A Pitch is composed of an Accidental object. Both Stream and DefinedContexts objects use object aggregation; they are not responsible for maintaining the life of their components. Streams, however, use references; DefinedContexts use weak references. Thus, the Stream establishes that the Note is a member of the Stream; the DefinedContexts establishes the specific offsets for the Note within one or more Streams. Further, a Stream can be deleted and garbage collected regardless of its presence in a DefinedContexts object, and without deleting its components (assuming those components are referenced elsewhere).

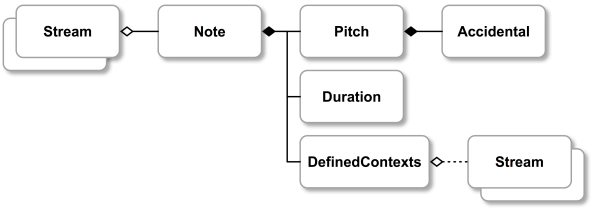

Figure 2. Object composition and aggregation using normal and weak references.

Figure 3 demonstates the basic functionality of Streams and other Music21Objects, including creating Note and Metadata objects and positioning Stream subclasses (Measure, Part, and Score) within other Streams. As much as possible, Streams mimic Python list functionality, with added support for handling offsets and durations. Thus, the `append()` method adds an element at the sum of (1) the highest offset and (2) the duration of the element at that offset; when given elements in a list, they are appended in succession. The `insert()` method permits positioning an element anywhere in the Stream given an offset. The `index()`, `pop()`, and `remove()` methods function similarly to Python lists.

```
# create two half notes and a measure
n1 = note.Note('g3', type='half')
n2 = note.Note('d4', type='half')
cf1 = clef.AltoClef()
m1 = stream.Measure(number=1)
m1.append([n1, n2])
m1.insert(0, cf1)
# the measure has three elements
assert len(m1) == 3
# the Note's offset is the most-recently set
assert n2.offset == 2.0
# this Stream automatic sorts the Clef first
assert m1[0] == cf1
# list-like indices follow the sort order
assert m1.index(n2) == 2
# find an element based on a given offset
assert m1.getElementAtOrBefore(3.0) == n2

m2 = stream.Measure(number=2)
n3 = note.Note('g#3', quarterLength=0.5)
n4 = note.Note('d-4', quarterLength=3.5)
m2.append([n3, n4])
# n4's appended position is after n3
assert n4.offset == .5
assert m2.highestOffset == .5
# can access objects on elements, here n4
assert m2[1].duration.quarterLength == 3.5
# Stream duration is automatically set
assert m2.duration.quarterLength == 4

p1 = stream.Part()
p1.append([m1, m2])
# the part has 2 components
assert len(p1) == 2
assert p1.duration.quarterLength == 8
# access Notes from a Part with indices
assert p1[1][0].pitch.nameWithOctave == 'G#3'
# or do the same with musical terminology:
assert p1.measure(2).notes[0].name == 'G#'


s1 = stream.Score()
s1.append(p1)
```

```
md1 = metadata.Metadata(title='a score')
s1.insert(0, md1)
# calling show renders output
s1.show() # 'musicxml' is the default
```

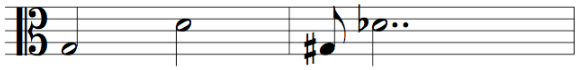

Figure 3. Appending, positioning, and getting values from Streams, as well as rendering MusicXML output.

Figure 4 diagrams the structure built in Figure 3, showing the hierarchical structure and the overlapping, timed events found within a single hierarchical level. Object names are those used in the Python script; offset positions are relative.

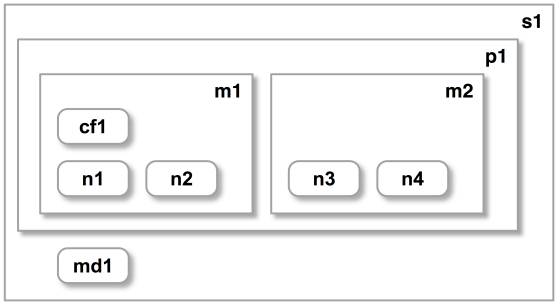

Figure 4. Common composition of music21 objects (e.g., n1 for a Note, cf1 for a Clef) and Stream subclasses (e.g., m1 for a Measure, p1 for a Part).

While Figure 3 builds a common hierarchical structure, such formations are not required in `music21`. When requesting an output format that necessitates measures, parts, and a score, for example, the Stream can build a copy with elements partitioned into default measures.

3.2. One Element, Multiple Containers

There are numerous conceptual and practical reasons why an element could simultaneously be in multiple independent containers, each with a different offset. For Music21Objects, the DefinedContexts object makes this possible. The DefinedContexts object can store any number of locations, each defined by a weak reference to a Stream and an offset. Thus, a single Note instance can simultaneously reside in multiple Streams at different offsets. Manipulations to the Note's pitch or duration in one Stream propagate to all Streams in which that Note is a member.

Presently, `music21` does not permit a Music21Object to have multiple locations in the same Stream, although the existing design will, with a few extensions, support this approach.

Continuing from the previous Python script, Figure 5 positions a single Note in multiple Streams and demonstrates using the `transpose()` method to transform the single instance in-place.

```
s2 = stream.Stream()
s3 = stream.Stream()
s2.insert(10, n2)
s3.insert(40, n2)
# last assigned offset
assert n2.offset == 40
# getting a location-specific offset
assert n2.getOffsetBySite(m1) == 2.0
assert n2.getOffsetBySite(s2) == 10
# the None site provides a default offset
assert n2.getSites() == [None, m1, s2, s3]
# the same instance is found in all Streams
assert m1.hasElement(n2) == True
assert s2.hasElement(n2) == True
assert s3.hasElement(n2) == True

# only offset is independent to each location
n2.pitch.transpose('-M2', inPlace=True)
assert s2[s2.index(n2)].nameWithOctave == 'C4'
assert s3[s3.index(n2)].nameWithOctave == 'C4'
assert m1[m1.index(n2)].nameWithOctave == 'C4'

# the transposition in the original context
s1.show('musicxml')
```

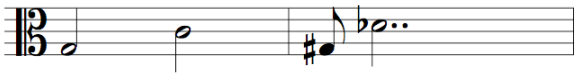

Figure 5. Manipulating a note in multiple containers

3.3. Simultaneous Access to Hierarchical and Flat Representations

Since a Stream can reside in another Stream, the offsets of a Stream's components are relative only to their immediate container. For example, a Measure might contain Notes at offsets 0 and 2; the Measure, however, might be positioned in a Part at offset 8. In order to determine the non-hierarchical, or flat, offset of the components, the offset of the container must be added to each component (resulting in offsets of 8 and 10 in this example), and this process must be done upward for all nested containers.

For symbolic music processing, simultaneously having both nested and flat structures, and both relative and non-relative offsets, is extremely important. For example, sometimes users may want to navigate through Notes, solely in the context of a Measure, to find out at what offset beat two occurs; at other times, users may want to navigate through Notes in the context of an entire Score to determine how far apart instances of a certain Chord are found.

Each Stream has a `flat` property that provides quick access to an independent, flattened representation of the Stream. In creating this representation, Music21Objects are not copied; they are simply inserted into a flat Stream according to the flat offset values, calculated as shown above. The flat Stream is sorted by the offset of each object. Figure 6 demonstrates the `flat` property, first with the objects created in the previous examples, and then by adding a second Part. The `getOffsetBySite()` method is used to return the offset for the Stream provided as an argument.

```
# lengths show the number of elements
s1Flat = s1.flat
assert len(s1) == 2
assert len(s1Flat) == 6
assert s1Flat[4] == n3
assert s1Flat[5] == n4
```

```
# adding another Part to the Score results in a
# different flat representation
n5 = note.Note('a#1', quarterLength=2.5)
n6 = note.Note('b2', quarterLength=1.5)
m3 = stream.Measure(number=1)
m3.append([cf2, r1])

r1 = note.Rest(type='whole')
cf2 = m3.bestClef() # cf2 is a BassClef
m4 = stream.Measure(number=2)
m4.append([n5, n6])
p2 = stream.Part()
p2.append([m3, m4])
s1.insert(0, p2)

# objects are sorted by offset
s1Flat = s1.flat
assert len(s1) == 3
assert len(s1Flat) == 10
assert s1Flat[6] == n3
assert s1Flat[7] == n5
assert s1Flat[8] == n4
assert s1Flat[9] == n6

# the B in m. 2 (=p2,m4) now has offsets for
# both flat non-flat sites
assert n6.getOffsetBySite(m4) == 2.5
assert n6.getOffsetBySite(s1Flat) == 6.5

s1.show()
```

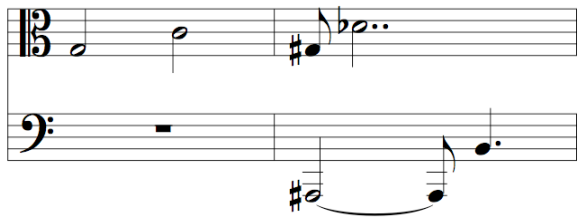

Figure 6. Access to flat and non-flat representations, and adding and displaying an additional Part.

3.4. Iterating and Filtering Elements by Class

The `music21` system makes great use of subclassing and class definition to distinguish system components. Each clef and each ornament type, for example, are given unique class definitions. As all Music21Objects can reside on Streams, it is often necessary to filter a Stream by class, return a new Stream with just the desired Music21Objects, and then process the results. As is typical in object-oriented languages, more general classes, inherited by subclasses, can be used for broader filters; specific class types or lists of classes can be used for more narrow filters.

For example, a complex polyphonic work might contain nested Streams defining Parts, Measures, and Voices, and within these Measures have Clef, TimeSignature, Note, Rest, and Chord objects. If a user needs to find the distribution of pitch usage, or look for a specific pitch, iterating over all these elements is unnecessary. Instead, the `getElementsByClass()` method on the flat representation can be used, where a list of desired classes (Python names or strings) is given. The returned, independent Stream includes only the matching classes with offset positions transferred from the source. As with Python lists, the Stream can be iterated in standard loop syntax.

Because calls to `getElementsByClass([note.Note, chord.Chord])` are so common, the `notes` property on Streams provides quick access to a new Stream containing only Note and note-like entities.

Figure 7 demonstrates applications of using `getElementsByClass()` and the `notes` property, iterating over various collected Music21Objects and gathering information or transforming objects.

```
# get the Clef object, and report its sign,
# from Measure 1
assert m1.getElementsByClass('Clef')[0].sign ==
'C'
# collect into a list the sign of all clefs in
# the flat Score
assert [cf.sign for cf in
s1.flat.getElementsByClass('Clef')] == ['C',
'F']

# collect the offsets of Measures in the first
# part
assert [e.offset for e in p1.elements] == [0.0,
4.0]
# collect the offsets of Notes in the first
# part after flattening
assert [e.offset for e in p1.flat.notes] ==
[0.0, 2.0, 4.0, 4.5]

# get all pitch names
match = []
for e in s1.flat.notes:
    match.append(e.pitch.nameWithOctave)
assert match == ['G3', 'C4', 'G#3', 'A#1', 'D-
4', 'B2']

# collect all the Notes using the
# getElementsByClass form and transpose them
# up a perfect fifth
for n in s1.flat.getElementsByClass('Note'):
    n.transpose('P5', inPlace=True)

s1.show()
```

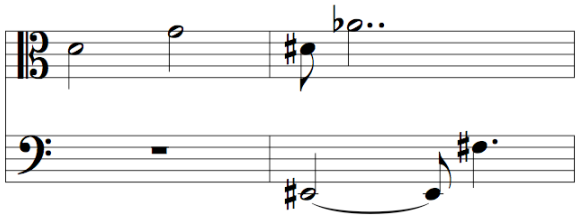

Figure 7. Iterating and filtering a Stream by class, and then displaying the results of transposition applied to filtered Note objects.

Similar to the Stream returned by the `flat` property, the Streams returned by `getElementsByClass()` and the `notes` property offer alternative "views" of the same Music21Components. Figure 8, expanding the previous presentations, illustrates references of the same Music21Objects in multiple parallel Stream instances, each retaining relative but independent offset positions.

With the frequent derivation of one Stream from another, by usage of both the `flat` property as well as by `getElementsByClass()` and similar methds, it is useful

to store and allow access to the chain of previous Stream forms from which the current Stream is derived The `derivesFrom`, `derivationChain`, and `rootDerivation` properties of Streams permit accessing the immediate ancestor, the complete list of ancestors, and the oldest ancestor, respectively. In addition, Music21Objects have a `derivationHierarchy` property that builds a list of containers moving upward recursively through the Music21Object's `activeSite`. For instance, a transposed Measure object might derive from an existing Measure object that is contained in a Part. The `derivationHierarchy` lets the transposed Measure get information from that Part.

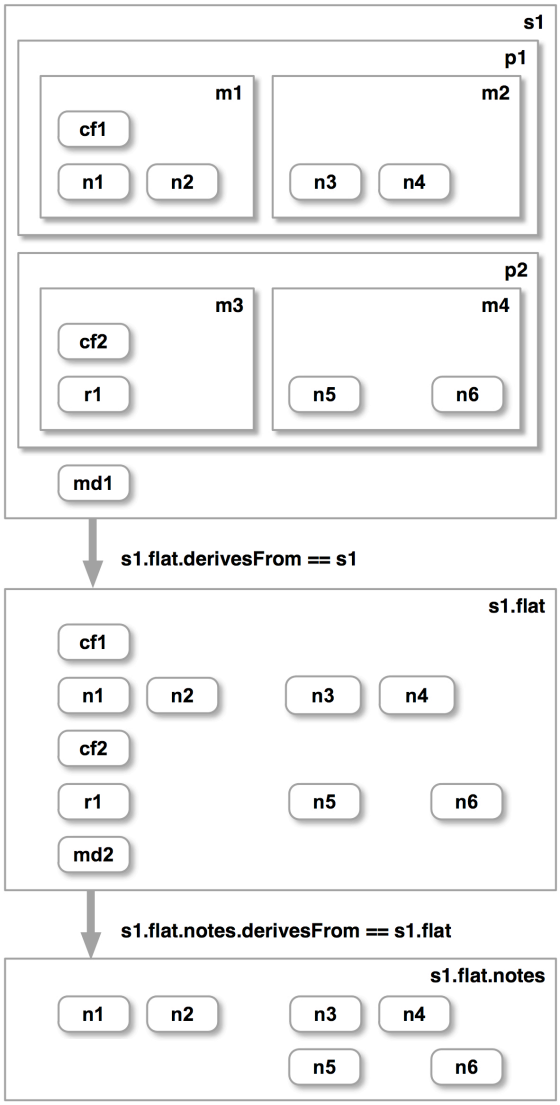

Figure 8. Simultaneous representation, and procedural derivation, of the same music21 object instances (e.g., n1 for a Note, cf1 for a Clef) in multiple, parallel Streams.

3.5. Searching Locations by Context

One reason notation-based music representations pose particular challenges to software processing is due to the frequent use of incomplete, partial, or context-dependent symbols and specifications. Clefs and time signatures are good examples. A clef, specified in the first measure, is applicable for all subsequent measures, unless a new clef appears. As with many musical representations, `music21` stores Clef and TimeSignature objects in Measure objects, and does not require these to be needlessly repeated in every measure. `Music21`, however, offers powerful tools for elements to search for classes, either upward, through containers, or up-stream, searching for most recently-encountered objects. Rather than always having to serially process elements in order, isolated Music21Objects can discover their context.

The `getContextByClass()` method, available on all Music21Objects, will first search the `activeSite` for the specified classes. If not found, `getContextByClass()` is called on each container for each location, and each location's location, recursively until a match is found. The search process uses `getElementAtOrBefore()` to return the first matched class found at or before the offset of the caller, with offset position determined relative to the appropriate container. Thus, a note in Measure 8 can find a Clef change in Measure 2 by first searching its Measure, then searching the flat Part that contains that Measure. This mechanism is a sort of backtracking hierarchical search.

Figure 9 demonstrates how `getContextByClass()` finds the Clefs and Measures most relevant to a given Note from a flat Stream. Importantly, the Note can find the appropriate Clef even when the Note is moved into a Stream not stored in the overall Score hierarchy. As the DefinedContexts object stores all locations, including locations in flat Streams, additional parameters, such as `sortByCreationTime='reverse'`, need to be provided to force searching oldest locations first.

```
# a Note can always find its Clef
assert n4.getContextByClass('Clef') == cf1

# all Notes can find their Measure numbers
# even from a flat Score
match = []
for e in s1.flat.notes:
    match.append([e.name,
        e.getContextByClass('Measure').number])
assert match == [['D', 1], ['G', 1], ['D#', 2],
['E#', 2], ['A-', 2], ['F#', 2]]
```

Figure 9. Searching Locations by Context

3.6. Spanners and Non-Hierarchical Object Associations

As stated previously, it is an error to force all object arrangements into strict hierarchical groupings. Some musical elements, particularly notational elements, need to be represented solely as associations between other elements that may or may not be in the same, nested, or adjacent containers. Examples include slurs, dynamic wedge symbols, extended trill marks, piano pedal indications, and even staff groups. Some systems model such objects with start and end tags embedded in a strictly hierarchical structure (e.g., MusicXML), yet

such approaches require searching, from any member, backward or forward to find the associated objects.

`Music21` borrows the term "spanner" from Lilypond [9] and, in part, from the "Leaf Container Spanner" model employed in Abjad [1]. Spanners are Music21Objects that store a collection of spanned objects, called components, that can exist in any hierarchical or non-hierarchical arrangment.

Internally, the Spanner stores components in a specialized Stream subclass called SpannerStorage. This object, unlike other Streams, stores a reference to the Spanner within which it is instantiated. The Spanner object provides the interface to SpannerStorage, as typical Stream functionality may not always have the same meaning when applied to storing Spanner components. For example, the offsets of components in a SpannerStorage object are irrelevant. However, by storing components in a Stream subclass, elements can directly list all Spanners in which they are components. The Music21Object method `getSpannerSites()` returns all Spanners for which the object is a component, by simply looking at the DefinedContexts object for SpannerStorage locations, and returning the reference to the Spanner instance stored in the found SpannerStorage object.

The Python example in Figure 10 demonstrates basic functionality of Slurs, a Spanner subclass. The `getOffsetSpanBySite()` and `getDurationSpanBySite()` methods can be used to determine the relative offset and duration span of stored components. These methods require a site argument, as a Spanner might connect Music21Objects in different sites. A common site, usually a flat representation, must exist to find offset and duration spans.

```
# Spanners positioned in a Part and a Measure
sp1 = spanner.Slur([n1, n4])
p1.append(sp1)
sp2 = spanner.Slur([n5, n6])
m4.insert(0, sp2)

# Elements can get their Spanners
assert n1.getSpannerSites() == [sp1]
assert n6.getSpannerSites() == [sp2]

p1Flat = p1.flat
assert sp1.getDurationSpanBySite(p1Flat) ==
[0.0, 8.0]
p2Flat = p2.flat
assert sp2.getDurationSpanBySite(p2Flat) ==
[4.0, 8.0]

s1.show()
```

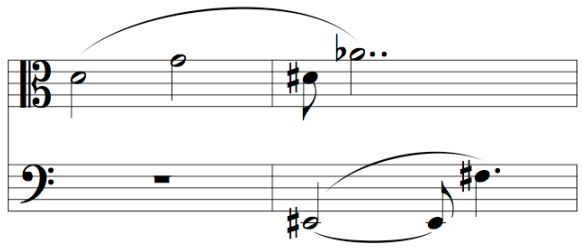

Figure 10. Creating a Slurs across Measure boundaries and realizing their duration span.

4. CASE STUDY

The following demonstration (Figure 11) illustrates the `music21` Stream model's utility in a more comprehensive example. This example is a typical application of `music21` in the following ways: (1) the code is compact and highly readable; (2) non-sounding notations can be examined with the same ease as sounding events; (3) a general approach that works for one work in one encoding can easily be employed on thousands of works in many encodings; (4) numerical output, textual output, and annotated score-based representations can all be created simultaneously. No other system for symbolic music manipulation offers all these features in such an easy-to-use framework.

The example below analyzes the beat location and the initial pitches of all melismas, that is syllables spread over multiple notes, in a score, and marks them with slurs. The example first parses a MusicXML representation of a fifteenth-century *Gloria* by D. Luca, stored in the integrated `music21` corpus (a collection of works distributed with `music21` for immediate experimentation and research). Using a flat representation of an extracted sub-section of Measures, the code finds all melismas by looking at pairs of Notes and determining the spans of Notes after a lyric ends and before a new Note starts. The code then spans each of them with a Slur. These slurs are then collected and are used to find the starting pitch and starting beat of the melisma (each one in this example happens on the downbeat), as well as the total duration of each melisma. The starting pitch and total duration are then printed.

Streams are critical for this procedure because: (1) a single Part, and a sub-section of Measures within that part, can be extracted while retaining musical and notational coherency (appropriate clefs, meters, etc.); (2) flat and filtered representations of the same Notes can be iterated over in series to examine pairwise relationships without altering the hierarchical representation; (3) elements stored in alternate representations can be modified, producing changes that are retained in the source representation; (4) the list-like functionality of Streams gives easy access to boundary elements.

```
nStart = None; nEnd = None
ex = corpus.parseWork(
    'luca/gloria').parts['cantus'].measures(
    1,11)
exFlatNotes = ex.flat.notes
nLast = exFlatNotes[-1]

for i, n in enumerate(exFlatNotes):
    if i < len(exFlatNotes) - 1:
        nNext = exFlatNotes[i+1]
    else: continue

    if n.hasLyrics():
        nStart = n
    # if next is a begin, then this is an end
    elif (nStart is not None and
    nNext.hasLyrics() and n.tie is None):
        nEnd = n
    elif nNext is nLast:
        nEnd = n
```

```
    if nStart is not None and nEnd is not None:
        nStart.addLyric(nStart.beatStr)
        ex.insert(spanner.Slur(nStart, nEnd))
        nStart = None; nEnd = None

for sp in ex.spanners:
    dur = sp.getDurationBySite(exFlatNotes)
    n = sp.getFirst()
    print(n.nameWithOctave, dur.quarterLength)

ex.show()
```

Printed Output:

```
('D5', 2.0)
('G4', 3.0)
('D5', 3.0)
('A4', 3.0)
('E4', 2.0)
('C4', 3.0)
```

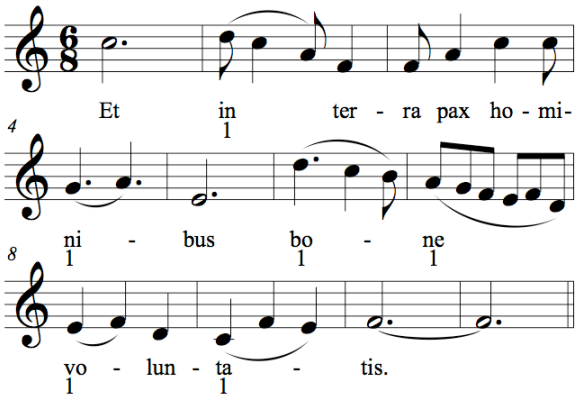

Figure 11. Counting, analyzing, and annotating melisma length. The printed output shows the starting pitch and melisma duration; the notated output shows the automated addition of slurs over melismas and the starting beat (in this case, always 1).

5. FUTURE WORK

The Stream model, as a critical component of the `music21` system, has undergone numerous revisions, expansions, and performance studies and optimizations. While it is mature, it might benefit from further refinements.

Presently, Spanners reside in Streams, unattached to objects. It is possible that optionally attaching Spanners to Music21Objects may offer organizational benefits. Also, as stated above, presently the same Music21Object instance cannot have multiple locations in the same Stream. This limit is in place for system simplicity, but, if removed, might offer a valuable increase in functionality.

6. ACKNOWLEDGEMENTS

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

7. REFERENCES

[1] Bača, T. and V. Adán. 2007. "Cuepatlahto and Lascaux: two approaches to the formalized control of musical score." Available online at http://www.victoradan.net/media/texts/cuepatlahto-and-lascaux_.pdf.

[2] Buxton, W. 1978. Design Issues in the Foundation of a Computer-Based Tool for Music Composition. Toronto: Technical Report Computer Systems Research Group.

[3] 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): pp. 10–20.

[4] Cuthbert, M. S., and C. Ariza. 2010. "`music21`: A Toolkit for Computer-Aided Musicology and Symbolic Music Data." *Proceedings of the International Society on Music Information Retrieval*, pp. 637–42.

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

[6] Hewlett, W. B. 1997. "Musedata: Multipurpose Representation." In E. Selfridge-Field, ed. *Beyond MIDI: the Handbook of Musical Codes*. Cambrdige: MIT Press, pp. 402–447.

[7] Hiller, L. 1981. "Composing with Computers: A Progress Report." *Computer Music Journal* 5(4): pp. 7–21.

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

[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] Oppenheim, D. V. 1989. "Dmix: An Environment for Composition." *Proceedings of the International Computer Music Conference*. San Francisco: International Computer Music Association, pp. 226–233.

[11] Oppenheim, I. and C. Walshaw, J. Atchley. 2010. "The abc standard 2.0." Available online at http://abcnotation.com/wiki/abc:standard:v2.0.

[12] Pope, S. T. 1996. "Object-oriented music representation." *Organised Sound* 1(1): pp. 56–68.

[13] Selfridge-Field, E. 1997a. "Beyond Codes: Issues in Musical Representation." In E. Selfridge-Field, ed. *Beyond MIDI: the Handbook of Musical Codes*. Cambrdige: MIT Press, pp. 565–572.

[14] Selfridge-Field, E. 1997b. "Introduction: Describing Musical Information." In E. Selfridge-Field, ed. *Beyond MIDI: the Handbook of Musical Codes*. Cambridge: MIT Press, pp. 3–38.