UNIVERSITY OF CALIFORNIA, SAN DIEGO

denm (dynamic environmental notation for music): A Performance-centric Music Notation Interface

A Thesis submitted in partial satisfaction of the
requirements for the degree
Master of Arts

in

Music

by

James Bean

Committee in charge:

Professor Rand Steiger, Chair
Professor Miller Puckette
Professor Scott Klemmer

2015

The Thesis of James Bean is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

_____

_____

_____
Chair

University of California, San Diego

2015

DEDICATION

To my family and Pia.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF SUPPLEMENTARY FILES

Audio

1: Proof of Concept: *deserving of Songs* for two flutes and two contrabasses.

2: Field Test 1: *dry;run.* for flute, viola, and clarinet.

3: *ligament at distance* for solo flute

Scores

1: Proof of Concept: *deserving of Songs* for two flutes and two contrabasses.

2: Field Test 1: *dry;run.* for flute, viola, and clarinet.

3: *ligament at distance* for solo flute

ACKNOWLEDGEMENTS

Thank you to the performers, organizers, and production staff who create the overwhelming amount of concerts and events in the UC San Diego music department, to those in the Composition department who served as advocates and advisors during my multi-year detour away from and back to composing, and to my colleagues who helped me through the rich though challenging subjourneys I have confronted along the way.

VITA

2013	B.M. in Music Composition, University of Oregon
2015	M.A. in Music Composition, University of California, San Diego

PUBLICATIONS

James Bean, "**denm** (dynamic environmental notation for music: Introducing a performance-centric music notation interface)", *Proceedings of the First International Conference on Technologies for Music Notation and Representation - TENOR2015*, 74-80, Paris, France, 2015. Institut de Recherche en Musicologie.

ABSTRACT OF THE THESIS

denm (dynamic environmental notation for music): A Performance-centric Music Notation Interface

by

James Bean

Master of Arts in Music

University of California, San Diego, 2015

Professor Rand Steiger, Chair

In this thesis, I describe the state of development for an automatic music notation generator and tablet-based graphical user interface. The programs currently available for the automatic generation of music notation are focused on the compositional and theoretical aspects of the music-making process. **denm** (dynamic environmental notation for music) is being designed to provide tools for the rehearsal and performance of contemporary music. All of the strategies underlying these tools are utilized by performers today. These strategies traditionally involve the re-notation of aspects of a musical score by hand, the process of which can be detrimentally time-consuming. Much of what performers re-notate into their parts is composed of information latent in the musical model—the musical model

which is already being represented graphically as the musical score. **denm** will provide this latent information instantaneously to performers with a real-time music notation generator. The thesis presents a few of the techniques currently used by performers of contemporary music, the features with **denm** that are designed to make these techniques more efficient, as well as the underlying algorithms that make this possible.

Part I

Overview

denm is a software application for iPad tablet computers that automatically generates fully formatted music notation, with support for the notational techniques used by composers of contemporary music. In addition to generating music notation, **denm** generates supplemental graphical information that aids the learning, rehearsal, and performance processes of performers of contemporary music.

Performers often manually annotate supplemental information in their parts. Much of this information already exists in the musical model that was used to generate their part. Inspired by what performers have notated in their parts while learning and performing my own music, **denm** seeks to automate some of the tasks of performers that are detrimentally time-consuming. With a notational environment that can be tailored by a performer to the music that they are playing, I believe that performers will be able to make more informed interpretations of the music of today.

In Chapter 1, we will investigate some of the re-notational procedures of performers of contemporary music. Then in Chapter 2, we will look at the ways in which **denm** seeks to make these extant practices more efficient, freeing up time and energy for performers to focus on more substantive musical issues.

Chapter 1

Annotation Strategies of Contemporary Music Performers

Percussionist Steven Schick [1] explains the multiple facets of the learning, rehearsing, and performing processes well:

> Painted in broad strokes, it seems to me that the act of learning a piece is primarily one of simplification, while the art of performance is one of (re)complexifying. In the learning process, rhythms must be calculated and reduced to some potable form, the turbulence of the microforces of form must be generalized, and various kinds of inane mnemonics must be employed simply to remember what to do next. An artificial skin of practical considerations must be stretched tightly across the lumps of a living, breathing piece. Performance reinflates the piece, fine tuning its formal gyroscope, revivifying polyphonic structures, and packaging the intellectual energy of the score into meaningful physicality.

While learning and rehearsing a musical work, performers often add markings to the score from which they are performing. Extra notational information is often helpful for the performer to understand the structures of the music that they are playing. These markings are not the musical material itself, but rather supplemental information that improves a performer's understanding of a piece of music.

1.1 Color Coding

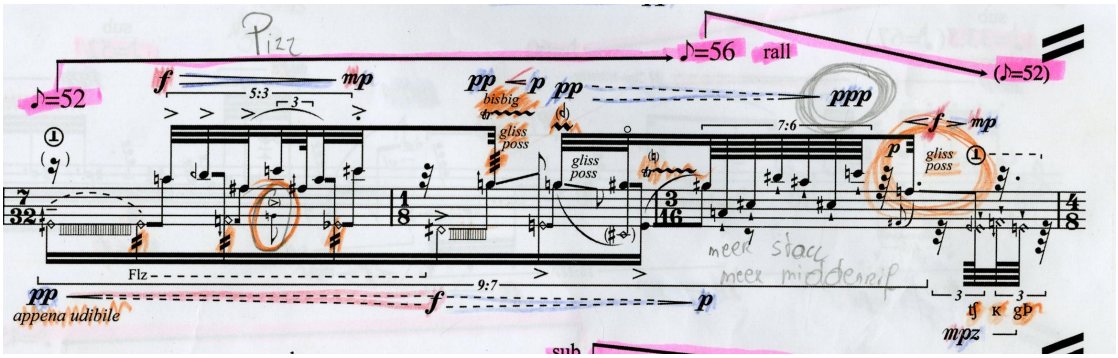

Figure 1.1: Performer-highlighted objects.

In Fig. 1.1, flutist Ine Vanoeveren highlights a variety of music notational objects with a system of colors and shapes to help clarify the learning of Brian Ferneyhough's *Sisyphus Redux* [2]. Vanoeveren stratifies different classes of musical objects by using different colors. As a result, very complex notational relationships are understandable at a glance.

The eye of a performer flickers quickly between many music notational objects in the act of performing a piece of music from a score. Complexly notated works make this process even more unreliable when utilizing traditional black-ink-on-paper notation preparation methods. As the amount of types of musical symbols on the page increases, it becomes more difficult for the performer's eye to "land" on the graphical representation of a particular stratum of musical information, in the process of evaluating the stratum's current state or value. When performers annotate their scores, they most often use different colors and translucencies of writing utensil to make more apparent the differentiations between the strata of musical information on the page.

In the case of Fig, 1.1, Vanoeveren highlights information about performing techniques (e.g., trill, bisbigliando, voiced sounds, fluttertongue, etc.) in orange colored pencil. Objects representing tempo and its fluctuations (e.g., tempo marking: eighth note = 52, rallentando from eighth note = 56 to eighth note = 52) are highlighted in pink marker. Lastly, dynamics are highlighted with red and blue

colored pencil: dynamic markings and interpolations highlighted with red colored pencil are ***mf*** or louder, while those highlighted with blue colored pencil are ***mp*** or quieter.

Without the clarifying markings made by Vanoeveren, this score may be quite difficult to parse, particularly in real-time. Many of the compositional strategies and performing techniques in this work are not traditional, though they are presented with conventional notational techniques that have a low headroom for notational complexity. This music bends these traditional notational techniques to the point of breaking, resulting in a saturated notational image. The dynamic and highly differentiated nature of the musical model of this work is neutralized to flat black graphical representation.

1.2 Ensemble Cues

When musicians perform in groups from a notated score, each performer performs from an abbreviated form of the full score, called a Part. A Part contains only the single viewing performer's music, omitting the music of the other instruments in the ensemble. In Parts for orchestral music, where a given performer may have many rests between passages, an additional staff of music shows supplemental musical information of another player that serves as a Cue.

By carefully curating this environmental information, the notational image for a single performer is able to be concise, while still making providing the performer with enough musical information to remain oriented within the ensemble and the structure of the composition. Cues may show all of the musical information that the other performer may see (e.g., pitch, rhythm, dynamics, articulations, etc.), or they may show only an abbreviated combination of this musical information (e.g., only rhythm). The decision to show some or all of the musical information to a performer is defined by the musical context. In traditional music preparation, the preparer of the Parts (the composer, or a dedicated engraver) has agency in this decision. In some contexts, this may be sufficient; however, in more contemporary compositional contexts, where the complexity of ensemble

relationships may be higher, such a static and centralized distribution of agency in notational representation can be inefficient (i.e., the knowledge of different aspects of an ensemble environment are useful at different points in the rehearsal and performing processes).

In small ensemble contexts, it is common for performers to manually notate varying amounts of detail of other performers' music in their Part. Performers may notate just the rhythmic information, as is the case of Tyler J. Borden's annotations on his part of *Thoracic Asphyxiating Dystrophy* by Harris Papatrechas [3]. In other cases, they may notate other information (e.g., pitch, dynamics, articulations, etc.) as is appropriate for a given context. By making explicit the musical relationships that a performer has with the other performers, a performer may adjust their interpretation of their Part to make these musical relationships most effective. Further, the annotations of their colleague's Parts enables performers to glance ahead to an upcoming ensemble relationship, in order to modify their current performative trajectory to this future point. Larger-scale ensemble relationships can then become effective to an audience, as they become explicit to the performers thoughtfully embodying the composition.

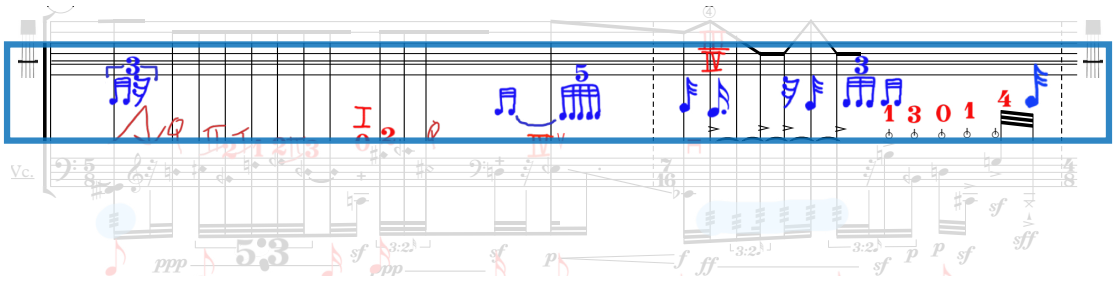

Figure 1.2: Performer-notated cues from ensemble.

In Fig. 1.2, cellist Tyler J. Borden manually notates the rhythmic information of the clarinetist in the cello and clarinet duo by Papatrechas. These annotations have been made with the ForScore application [4] for iPad tablet devices. This program displays PDF (Portable Document Format) representations of music notation, allowing performers to add annotations directly onto the score by interacting with the touch screen element of the iPad.

1.3 Meter

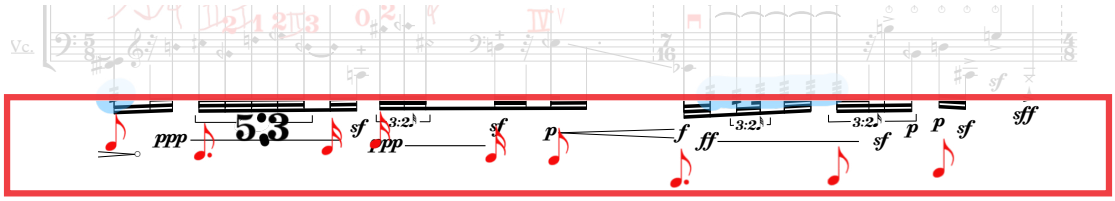

Figure 1.3: Performer-notated clarification of meter.

Borden notates a clarification of the underlying meter of the passage shown in Fig. 1.3 for several reasons: the rhythmic environment is inherently complex, the underlying meter of this passage is a series of uneven durational values with varying subdivision values, and the notated horizontal spacing of the music in this passage is irregular. For example, the third, fourth, and fifth performer-notated rhythmical values have the same durations, though the horizontal space occupied by each varies drastically.

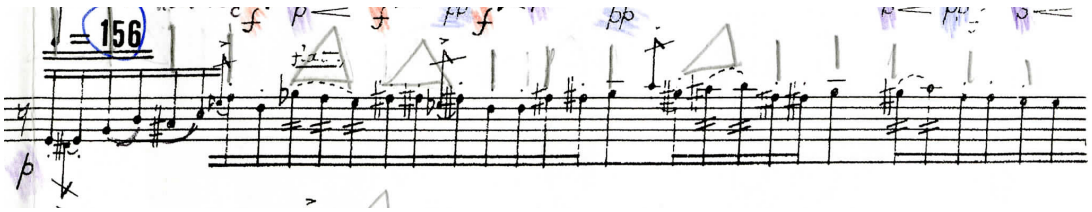

Figure 1.4: Performer-notated clarification of meter.

Vanoeveren uses vertical lines and triangles to intelligently clump note events in Franco Donatoni's *Nidi* [5], shown in Fig. 1.4. Without these notations, it would be easy to get lost in the unmetered collection of notes that comprise this work. The vertical lines represent duple-beats, while triangles represent triple-beats. The MetricalAnalyzer class in **denm** (described in Sec. 10.1) uses the organization of beats into duple- and triple-beat groupings as a model of representing an underlying metrical structure for rhythms of arbitrary complexity. Vanoeveren's process elucidates the natural tendency to group rhythmic events for the process of memory.

Click-tracks

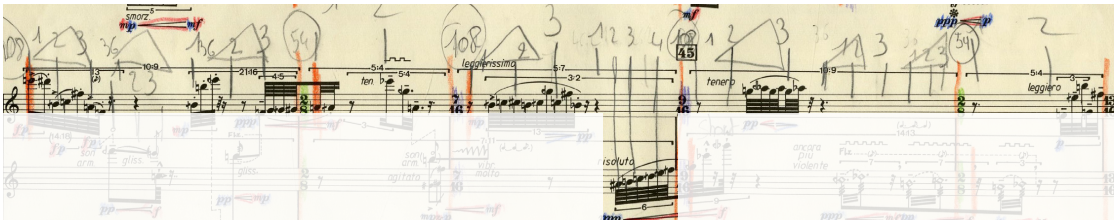

Figure 1.5: Performer-notated clarification of meter.

Performing *Mnemosyne* by Brian Ferneyhough [6] necessitates a significant amount of rehearsal technique infrastructure. This work, an excerpt shown of which is shown in Fig. 1.5, is written for solo bass flute, along with a recording of eight bass flutes. The live performer synchronizes with the recording with the help of a "click-track" (i.e., a metronome specifically tailored to the meter of a piece). This "click-track" is critical to the learning, rehearsing, and performing of this work as the metrical profile is complex. As a result, performers add more annotations relating to the meter than they may in other contexts.

There are several aspects of this Ferneyhough work that make it intrinsically difficult with respect to time. The metrical structure does not only contain subdivision values that are results of the statement 2^n (e.g., 8, 16, 128, etc. — the case in most traditionally notated Western music). Instead, meters in this work incorporate another element of m into the equation, where subdivision values are results of the statement $m * 2^n$ (e.g., 10, 24, 56, etc.). The traditional subdivision idiom is a subset of values of the expanded model, resulting from the statement of $2 * 2^n$. In this case, $m = 2$, as the subdivision values are duple-beat constructs. In some meters in *Mnemosyne*, m may be 3, 5, or 7. In the cases where $m \neq 2$, the underlying pulse is a non-duple subdivision of the tempo (e.g., triplets, quintuplets, or septuplets).

Consequently, the "click-track" is very specific and must be understood thoroughly by the performer. Vanoeveren adds several layers of annotations in the score to clarify the grouping of beats within a meter, to keep synchronized with the "click-track". Vertical lines represent duple-beats and triangles represent

triple-beats, as in the Donatoni example shown in Fig. 1.4. Vanoeveren takes this method even further for the Ferneyhough, as she annotates the numerical representations of the internal beat groupings, in addition to the global pulse. For example, Vanoeveren decides that the $\frac{7}{16}$ measure shall be subdivided into subgroups of `[3,4]`. There are no events that actually fall on these subgroup points, though this has proved the most reliable scaffolding of meter for her. In addition to the duple-beat-representing lines and the triple-beat-representing triangles, Vanoeveren notates the numbers of each subdivided beat within the subgroups she has decided.

Chapter 2

Features of denm

denm seeks to provide many of the extant annotational strategies of performers that are described in Chapter 1 more efficiently. Often, what is annotated by performers onto their scores already exists symbolically within the musical model that is being used to generate their part. **denm** provides performers with the opportunity to use these strategies without spending as much time preparing them.

2.1 Color Coding

For concerns of cost and reliable reproducibility, music notation is most often printed with black ink on paper. This technique is sufficient for music notational contexts that have a low density of music notational information and few graphical objects. Many musics written today incorporate extended forms of notation that utilize music notational concepts beyond those found in traditional music notation. As the amount of music notational objects within a music notational context increases, the low headroom of the traditional paradigm is made apparent. As a result, performers spend a considerable amount of time and energy annotating their scores with colored markings, to make sense of a dense notational image.

As described in Sec. 1.1, performers are forced to work around the inefficiencies of the over-saturated black-ink music notational images. Within these

notational environments, it is very hard to establish contrast between classes of musical elements. When performing from a musical score, a performer's eye flickers between classes of music notational objects that have influence of varying scales of musical activity. This flickering is less agile when a high a density of monochromatic notational images is present; it is difficult to discern the function of a given object quickly. Subsequently, performers will overlay colored annotations over a musical score, in order to make clearer the functional contrasts between notational objects. An example of this can be seen in Fig. 1.1.

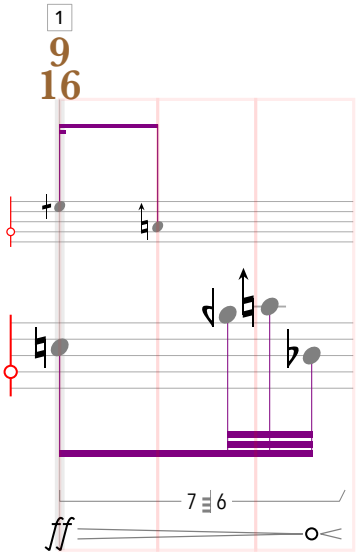

Figure 2.1: Different classes of music notational objects colored differently.

Graphical objects in **denm** are colored in order to create easily identifiable stratifications of function between different classes of music notational objects. Objects with lesser degrees of semantic influence on the objects around them (e.g., barlines, staff lines, etc.) are desaturated in color to be pushed toward the background of the music notational image. Objects that have direct semantic influence over objects around them (e.g., clefs and accidentals), though are a more rarely occurring, like clefs, have more highly saturated colors in order to be pushed toward the foreground of the music notational image. Certain objects, such as the light red rectangles shown in 2.1, are temporarily shown at the request of the user. These objects are positioned in the foreground, though are given a very low opacity. This

high degree of transparency allows these rectangles to be visible over the entire vertical span of a System, without occluding any other objects. See Part II for a more in-depth discussion of the topics of utilizing depth-of-field in contemporary music notation.

Many of the workarounds that performers use to overcome the over-saturated monochromatic environment of traditionally printed musical scores are unnecessary with sensible color coding of objects in a music notational environment. Certain strategies (e.g., color coding specific values of dynamic markings (shown in Fig. 1.1)) will be elective in future iterations of the software.

2.2 Ensemble Cues

Performers will often annotate varying levels of detail of other players' Parts in their own when playing in an ensemble. This process requires asking around the ensemble what each player is doing, in order to establish what to remember, and therefore annotate onto their Part.

The cues annotated onto performers' Parts are often representations of musical information latent in the musical model that is already being represented as the musical Score and Parts. **denm** makes it possible for a performer to show or hide any other players' part in an ensemble for any musical span. Further, any combination of musical components may be included or excluded, as desired by the performer. As in Fig. 1.2, merely notating the rhythms of the other player was the most efficient strategy for the cellist Borden.

In different circumstances, different information is helpful. In the case of orchestral cues, the engraver has control over what is curated for the performer. In most orchestral music, there are relatively few musical (and therefore notational) elements that a performer may be listening for. In the context of more complex chamber music, there may be many musical (and therefore notational) elements that could warrant attention (and therefore annotation).

In some cases, showing the pitches of another part may give a player an identifiable entity that helps in the orientation within the ensemble environment

and compositional structure (e.g., when salient melodies are present). In other cases, showing the pitches of another part may be helpful for tuning. In other cases, showing the pitches of another part may be superfluous. Most importantly, each of these cases can be applicable to the same musical passage, at different points in the learning, rehearsal, and performing processes. **denm** allows performers to curate their own music notational environment. Fig. 2.2 shows multiple representations of another performer's part in an ensemble context. Progressively less information is shown in each example. The fourth example shows the most spartan possibility, where only the points of events of the other part is shown.

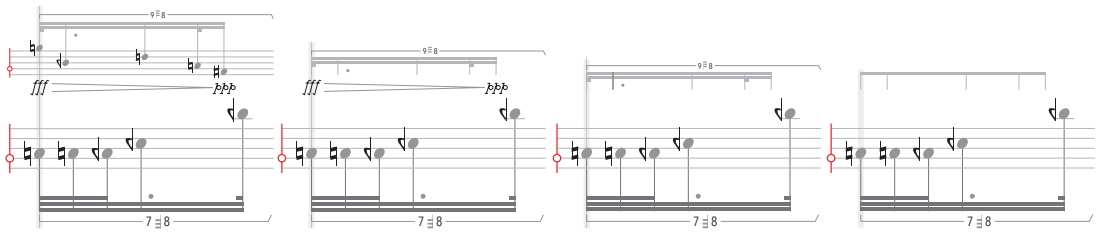

Figure 2.2: User-definable cues.

Rhythm Representation for Cues

When annotating rhythms of passages from other players, a performer may notate varying degrees of rhythmic specificity. In some cases, performers will notate lines with a horizontal relationship to their own part, and in other cases, annotating the full rhythm, with all of the beaming and tuplet information, is required for the cue to be useful.

denm incorporates a variety of strategies into a user-definable cue-ing system. In addition to showing any component of another player's part, the beaming and tuplet information (if present) can be optionally shown. The beaming (if a rhythm is represented as `metrical`) and the tuplet information (if a rhythm is represented as `numerical`) can be included according the user's preference for a given context. The first three examples in Fig. 2.2 show `metrical` and `numerical` representation of another player's rhythm, while the fourth example shows a rhythmic representation that is neither `metrical` nor `numerical`.

2.3 Meter

MetricalGrid

The first thing that many performers annotate into the part of a rhythmically context musical environment is a collection of vertical lines showing the beats of the measures [7]. This is often done hastily, in graphite, and vertically above the musical material. To accomplish this automatically via a dynamic screen representation of the score is simple, as long as the notational rendering engine has access to the underlying musical model at hand. **denm** allows the performer to show or hide a MetricalGrid, which is the extension of this vertical line technique, laid over the musical material. A MetricalGrid extends over an entire system of musical information. The lines comprising it are given a low opacity, allowing the musical information to show through. The lines extend to the bottom of the system, so that all events, regardless of vertical placement within a system, can be related visually to a global tactus.

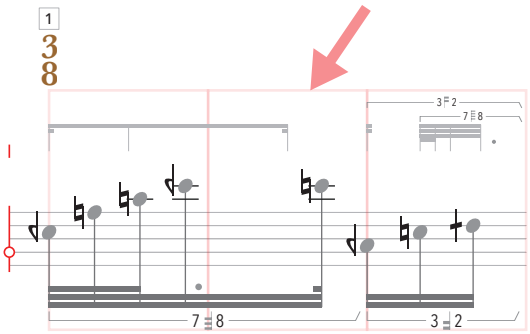

Figure 2.3: Simple Metrical Grid in use.

MetronomeGraphics

denm extends the chunking of durational values into groups of duple- and triple-beats that is naturally done by performers. MetronomeGraphics are the graphical representation of the metrical analysis process described in 10.1. In addition to operating on a Measure-level, as performers often do (see Fig. 1.4 and Fig. 1.5), MetronomeGraphics are applied to any container DurationNode in a Du-

rationNode tree structure (described in Sec. 7.4). In Fig. 2.4, an embedded tuplet rhythm is shown, where each tuplet bracket is given its own set of MetronomeGraphics. As you may see in the `11:12` tuplet bracket, there are two distinct levels of MetronomeGraphics. In this case, there are two nodes of MetronomeGraphics grouped, with the first node containing three "clicks", and the second containing two "clicks". An entire tree structure of prototype beat structures is created, with each container node containing two or three child nodes. The duple- and triple-beat paradigm is replicated recursively.

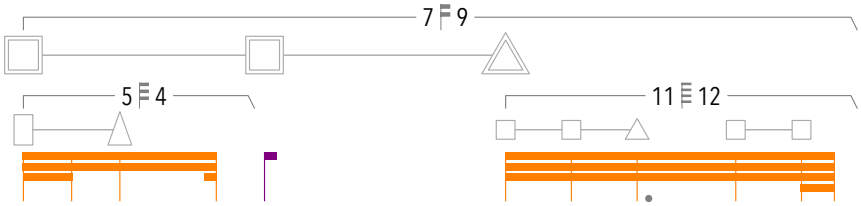

Figure 2.4: MetronomeGraphics in use.

Click-tracks

The underlying timing system is implemented to a proof-of-concept level within **denm**. This timing system provides a sample accurate (i.e., accurate to within $\frac{1}{44100}$ second) foundation for generating sonic and visual representations of metronomes tailored to the content of a musical work. When this "click-track" is fully implemented into the graphical dimension of the system, objects like the MetricalGrid and MetronomeGraphics will graphically represent what is heard sonically by the performer. A planned development will be to extend the hierarchical metrical analysis procedure (described in Sec. 10.1, currently graphically represented by MetronomeGraphics) into the sonic realm. That is, a hierarchy of beat-weightings will be sonically represented with different timbral construction of "clicks": strong-beats will have a certain sonic profile, while weaker-beats will have their own.

Part II

Graphic Design Concepts for Contemporary Music Notation

Performers interpret graphical representations of musical content, called scores. Composers today are most often the creators of these graphical representations. As such, there is an intertwining relationship between the musical content generated by a composer and the capabilities of the graphical representation of it: composers must create within the pragmatic boundaries afforded by the current state of music notational conception, while they may also stretch these pragmatic boundaries to fit their creative impulses. The cost to performers to learn new music notational concepts, the technical limitations of music preparation methods, and the effectiveness of new music notational concepts to represent desired musical contexts must be considered by a composer when experimenting with the extension of the shared music notational landscape. Performers of notated music are necessarily impacted by the music notational concepts represented in a score. As a result, the content of a notated musical composition is entangled within the methods of its graphical representation.

Chapter 3

Practical Considerations for Score Preparation By Example

Throughout the last four years, I have invested in the experimenting of and implementing different methods of preparing scores and parts. Many lessons have been learned regarding graphic design, and printing technology. How this all fits into the intricate process of preparing scores and parts is messy: I will go through a lineage of works, and will investigate the lessons learned by each. It is easy to see how the score-preparation strategies of each piece build upon those of the previous.

Starting in the fall of 2013, with Proof of Concept: *deserving of Songs* (described in Sec. 3.6), my creative focus became singularly aimed at authoring **denm** (dynamic environmental notation for music), a software application for iPad tablet computers. Projects of this nature contain many subprojects which are both implementations of techniques, and experiments of techniques at varying levels of development. Testing such things in a real-world situation is necessary, though it is often quite painful. Each subproject contains goals with shorter and longer aspirational trajectories embedded within it. While implementing any subproject, it is often quite difficult to point to an objective desired outcome, or even what is being "tested" at the current stage of the process. Upon reflection, the sequence of works presented here provides enough context to see some shorter-term and longer-term aspirational trajectories bearing fruit — enough at least to provide evidence that the current "tests", which are perhaps hard to define, will show their purpose

when the time is appropriate.

Perhaps it makes sense that many of the lessons learned the hard way pertain to the printing process: I am convinced now that the use of direct manipulation touch interfaces (e.g., iPads) will enable performers to access information about the music they are playing with greater efficiency than is currently possible with paper scores.

3.1 *This Will Be Changed and Made Solid I A*

This Will Be Changed and Made Solid I A for solo voice, is the first composition of mine that uses colored music notational objects in its score.If colors are used in musical scores, it is most often used statically to represent different strata of musical information (e.g., red objects belong to one category of information while green objects belong to another category). In this work, the mapping is within a single stratum, showing a continuous value within that stratum. Colors are mapped directly onto a two-dimensional organization of vowel space. As is indicated in Fig. 3.1, colors such as red, yellow, green, and blue represent corners of the IPA (International Phonetic Alphabet) diagram.

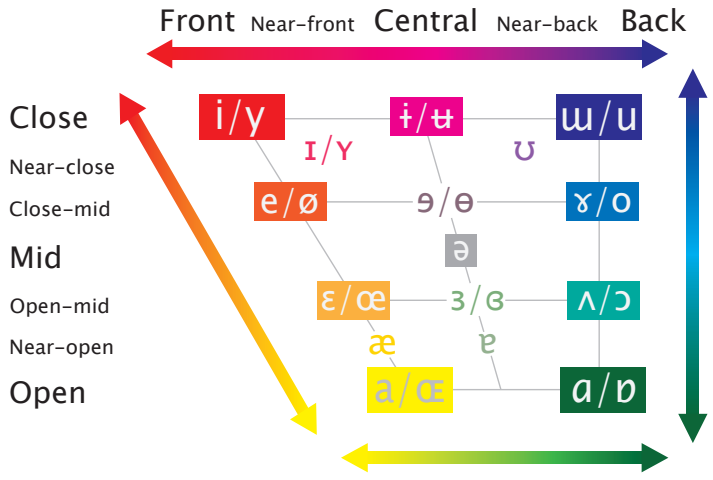

Figure 3.1: Two-dimensional color coding of vowel space.

This color mapping operates in two-dimensions: the height of the vowel, and

the backness of the vowel. The vowel shape of the performer shifts continuously between “aahs”, “uuhs”, “oohs”, “ees”, and between, as the colors representing these vowels does the same. The performer glides between these static vowel states continuously, as can be seen in Fig. 3.2.

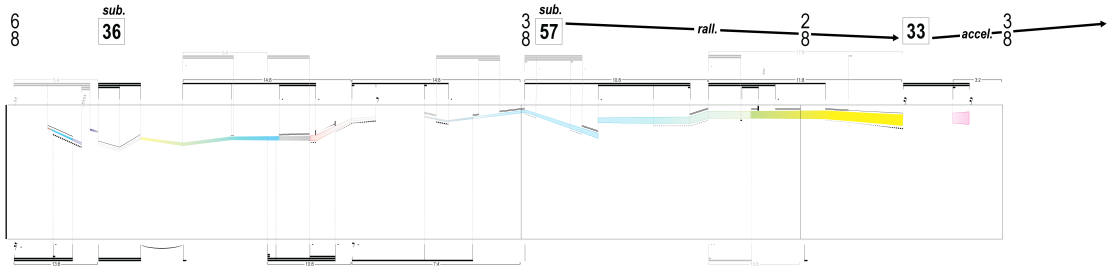

Figure 3.2: System view: *This Will Be Changed and Made Solid I A*.

In addition to a continuous definition of vowel placement in *This Will Be Changed and Made Solid I A*, there is a continuous definition of tongue placement and pressure. As described in Fig. 3.3, tongue placement and pressure is shown graphically with different stroking styles. The activity of the tongue is split into two parts: the tip of the tongue and the back of the tongue. The two zones of activity are notated in two places: the graphical information representing the tip of the tongue is placed above the vowel information, while the graphical information representing the back of the tongue is placed below the vowel information.

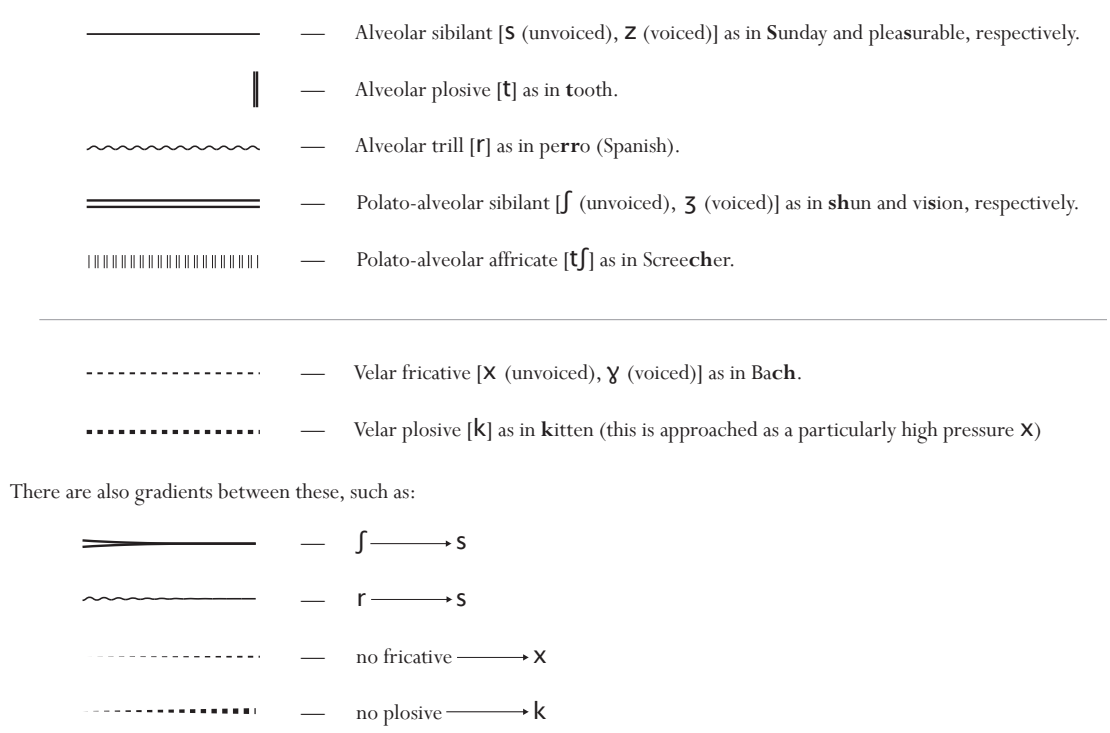

Figure 3.3: Explanation of tongue activity.

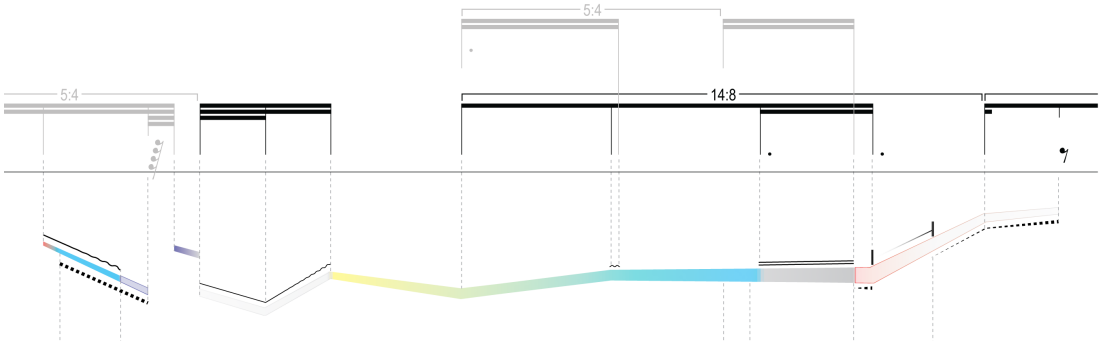

Figure 3.4: Detailed view: *This Will Be Changed and Made Solid I A.*

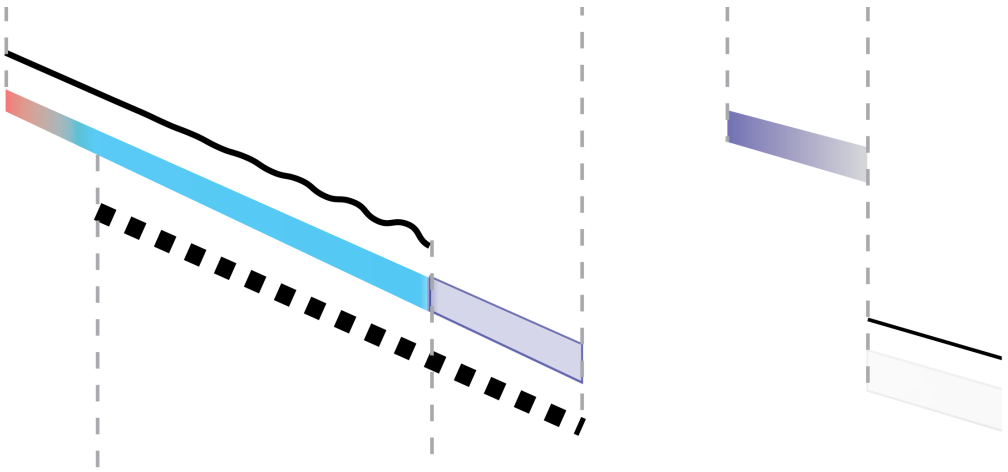

Figure 3.5: Very detailed view: *This Will Be Changed and Made Solid I A.*

Practical Considerations of Color Printing

Much of the musical information graphically represented in the score varies by subtle degrees. Slight shifts in tongue placement and vocal cavity shape define the character of the work from a musical perspective, but the graphical representations of it proved to be quite problematic in the printing process. Colors can be represented uniformly on high quality screen displays. As this work was built in Adobe Illustrator, the entire visual feedback process was screen-based. The cost of printing in color is quite high, and as a result, there were no printed proofs made to verify the success of the colors representing the continuous shifting of vowel shapes. While this work is successful on a screen (in a sense, the visual medium for which it was designed), there are certain issues revealed when the score is printed physically.

In addition to being costly, the color printing process can be unreliable. When printing a black and white image (as is the case with most music notation), a single mechanism carries black ink that applies ink to the page. However, when color printing is used, at least four different mechanisms carry different colors (most often Cyan, Magenta, Yellow, and Key (Black)). The printer must make multiple passes over the colored graphical image, applying varying amounts of each over the previous. This process, called "registration" is not perfect. With each pass, the

printer may be in slightly the wrong position. When this happens, a viewer is able to see fuzzy outlines around objects with a particular defining color (either cyan, magenta, or yellow). This artifact of color printing is called "misregistration" [8]. As music notational objects are quite small, the error tolerance of misregistration is significant in comparison to the size of the object.

The semantic information in the score of *This Will Be Changed and Made Solid I A* is carried in the color itself, not merely the shape or placement of objects. If the coloring system is unreliable, the notational environment is unreliable as a whole. When printing the score for this work, the coloring system was largely unreliable. Certain colors seemed to print more successfully: objects colored with cyan and magenta were the most reliable, even at very small sizes. Objects with other colors were blurry, and would disappear as soon as strokes became thin. When the printer is representing colors like cyan and magenta, it only makes a single pass with a single color; "misregistration" is no longer an issue. These colors print well, but others do not. There is a large discrepancy between the successfully and unsuccessfully printed colors. When a notational system is continuous (e.g., vowels shift continuously analogously with the colors), there must not be irregularities through the range of representation. The notational process of this work, while successful on high quality displays, presents challenges of irregularities in printing quality through its range representation.

3.2 *This Will Be Changed and Made Solid I B*

The color-related strategies of the score preparation process of *This Will Be Changed and Made Solid I B* for solo alto saxophone are directly related to the lessons learned in the printing process of *This Will Be Changed and Made Solid I A*. Colored objects in this work, most of which are thin strokes, are primarily magenta or cyan. The use of color is more conservative than the previous work, and the colors print consistently and successfully, contrasting well with the monochromatic conventional music notational objects set in the background of the notational image.

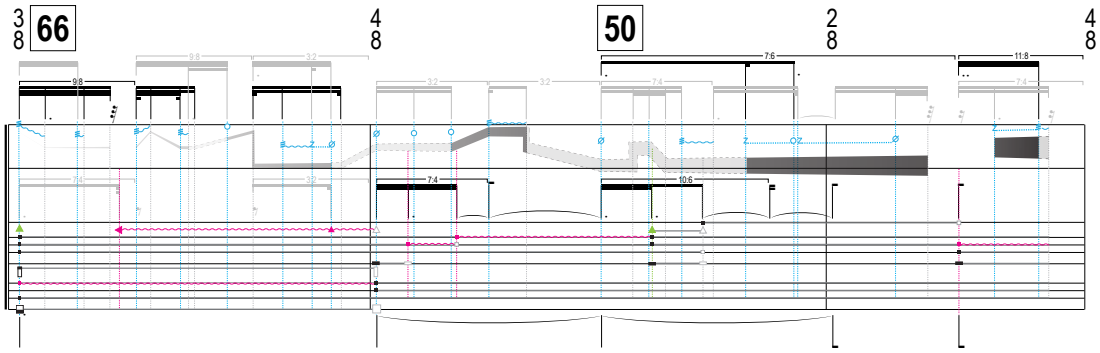

Figure 3.6: System view: *This Will Be Changed and Made Solid I B.*

The use of color in this piece is not continuous in the manner of *This Will Be Changed and Made Solid I A*. Different colors are used to differentiate strata of musical objects: cyan-colored objects represent the mouth-related activities of the saxophonist (e.g., articulation, tongue-techniques, airflow), while magenta-color objects represent dynamic components in the fingers of the saxophonist (e.g., trilled keys).

Practical Considerations of Grayscale Printing

While the use of color in *This Will Be Changed and Made Solid I B* is more conservative than that of the preceding work, new issues were found in the process of using grayscale in printing. In this work, there are two layers of rhythmic activity. To differentiate between these two levels, light gray and black are used. On the screen, a slight contrast between varying levels of darkness of grayscale, from white to black and all values in between, can be effective. When printing, the already-fragile music notational symbols are mode more fragile when grayscale values are used. The choices made intuitively for the darkness of gray rhythms here backfired: instead of an elegant graphical isolation of overlapping rhythms, half of the rhythms disappeared.

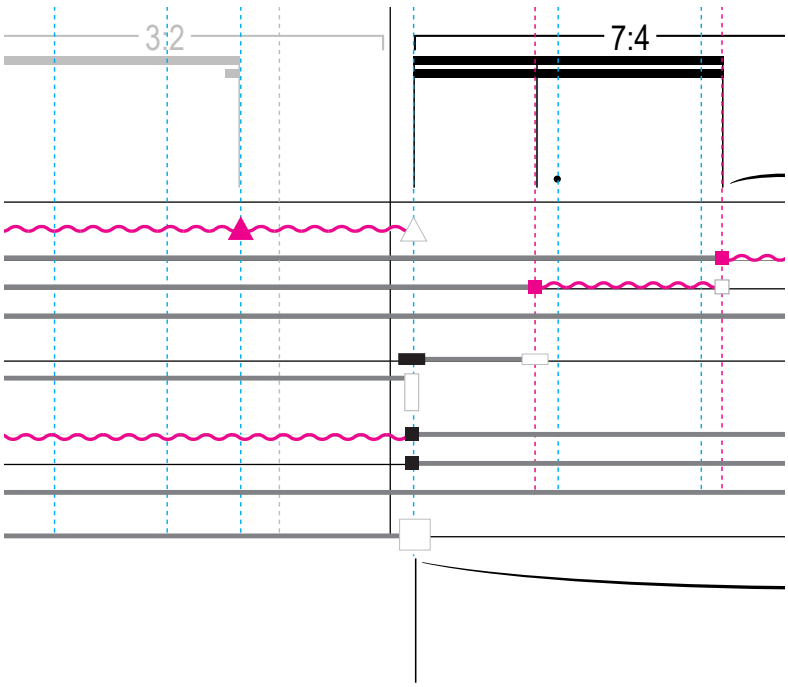

Figure 3.7: Detailed view: *This Will Be Changed and Made Solid I B.*

The process of producing grayscale colors in a print context is called "halftoning" [8]. Instead of having a dedicated ink for every possible darkness of gray, dots of black (or in some cases, combinations of CMYK colors) are printed in varying densities, proportionate to the darkness of the grayscale value. Light grays are not lighter inks, but they are rather a lesser amount of dark dots in the same area. As a result of this process, images with lighter gray values have a lower resolution. Small music notational objects (e.g., stems, small numbers), can become illegible when printed as a grayscale image, when black versions of the same objects will print effectively.

3.3 *This Will Be Changed and Made Solid II*

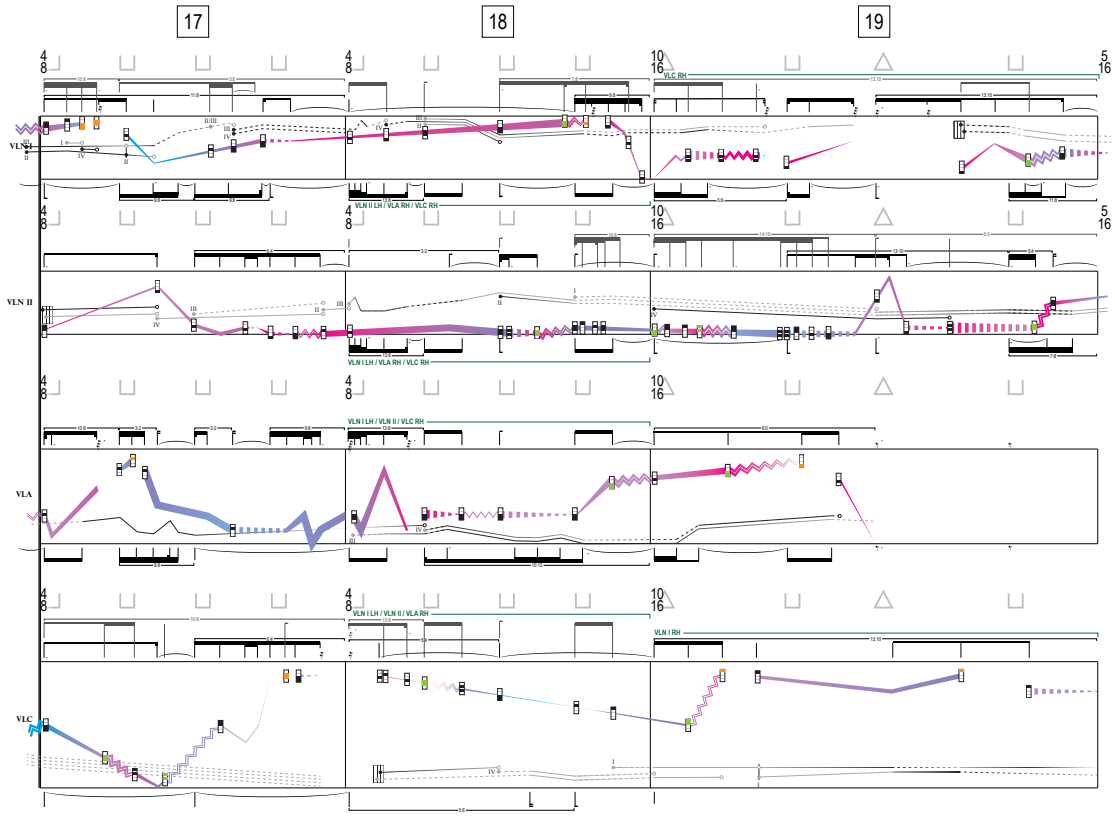

Figure 3.8: Page view: *This Will Be Changed and Made Solid II.*

This Will Be Changed and Made Solid II for string quartet refines the notation of a continuous mapping of a musical parameter to a color value. With several lessons learned from the previous score preparation processes, this score is significantly more successful. While *This Will Be Changed and Made Solid I A* utilizes an adventurous two-dimensional mapping of color space onto vowel space, the string quartet reduces the mapping of color onto a single physical dimension of movement (in this case, the placement of a string players bow along a string, between the nut and the bridge). Instead of exhausting the entire color space, the notation of the string quartet narrows the use of colors to the values between cyan (representing *molto sul tasto*: toward the nut) and magenta (representing *molto sul ponticello*: on the bridge). Colors that interpolate between these two extreme

color values represent positions between these two extreme positions.

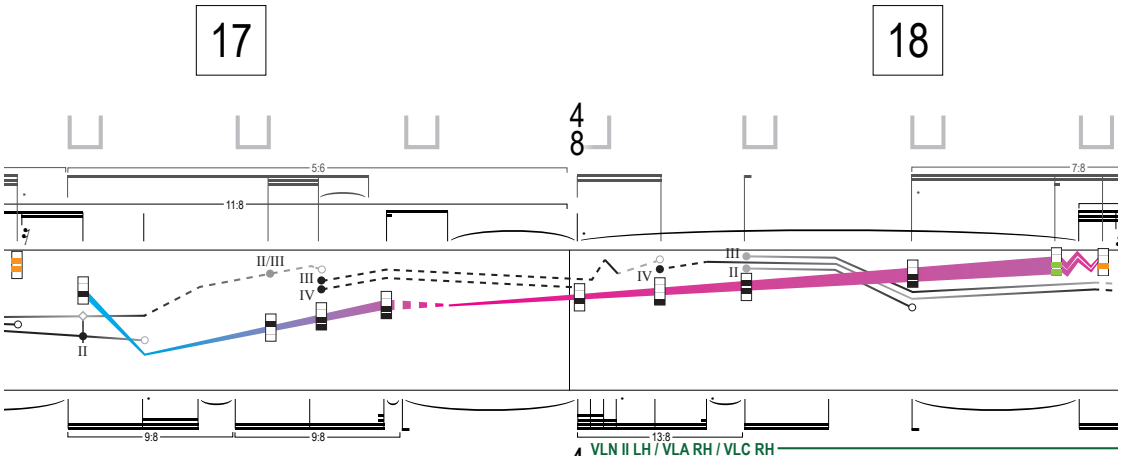

Figure 3.9: Violin part: *This Will Be Changed and Made Solid II.*

Colors like orange and green are used in specific circumstances. Orange and light green objects are always filled rectangles, which have an inherently robust characteristic, while dark green is saved for cue-ing information that is represented with capital case bold text and relatively thick lines. Varying levels of grays are used to indicate finger pressure, where conservative gray values and thick lines are used. Multiple layers of rhythmic information are also stratified in grayscale, as in *This Will Be Changed and Made Solid I B*, though more conservative (i.e., darker) gray values are used.

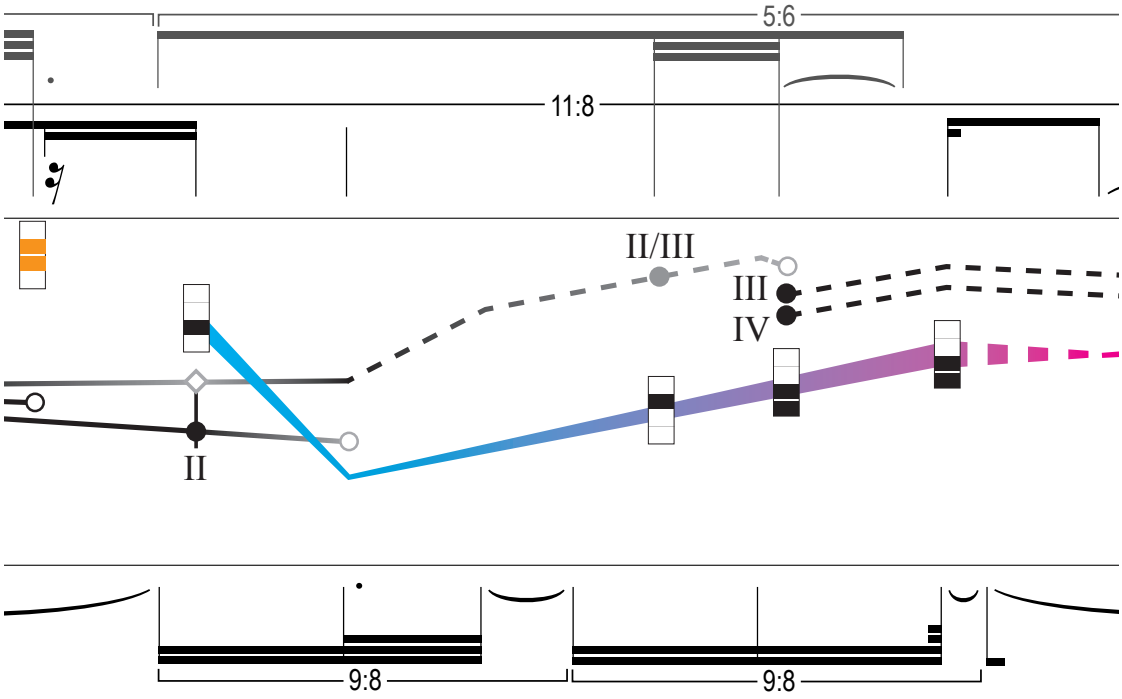

Figure 3.10: Detailed view: *This Will Be Changed and Made Solid II.*

The use of color in this score is successful, even though I was initially concerned of the misregistration issues faced in *This Will Be Changed and Made Solid I A*. Colors that have the potential to misregister are used here specifically in cases where either the semantic value of notational object is not reliant on the color (e.g., cue-ing text information, etc.) or where the graphical object itself is not intricate (e.g., filled rectangles).

More important than the technical success of printing colors is the utility of the colored notation in practice: does this color mapping make certain interpretive parsing tasks easier for the performer? Further, does this color mapping make certain tasks notatable that would otherwise not be? I found that the answer to both of these questions was *yes*. Not only does a continuous mapping of color afford an efficient way to notate continuous transitions in physical motion, this implementation of the color mapping still made it possible for the performer to begin objectively at the correct point (with a resolution of seven positions along the string). In my experience, the amount of time to land on the correct bow position with this notation is much less than with traditional methods of notation.

Conventionally, bow placement may be indicated in a score with a verbal description (e.g., *molto sul ponticello*, *m. sul pont.*, msp, etc.), and transitions between these objective states are notated with lines and arrows. Along with most black-ink-on-paper notational strategies, this sort of information has a low headroom for complexity. By embedding multiple strata of musical information into a single notational object, multiple layers of information can be embodied quickly by the performer. This is made possible only with the use of color.

Practical Considerations of Part Preparation

Each of the pieces demonstrated thus far were generated with a hybrid approach of using traditional music typesetting software (in this case, MakeMusic Finale), and a vector graphics authoring program (in this case, Adobe Illustrator). The traditional typesetting program is used to produce the layer of traditional musical information (e.g., measures, measure numbers, time signatures, rhythms, etc.), as this information is stored semantically in a subsystem within the typesetting application. This storing of the musical information enables the program to generate parts and a score: multiple views of the same musical model (this model is discussed in Sec. 6.1). The typesetting program automatically generates measure numbers, and places rhythms correctly, and allows an efficient interface for "flowing" systems (i.e., deciding how much music is distributed on a single system, and at what point music is flowed to the next system). However, traditional typesetting programs are categorically poor graphics editors.

The skeleton of pieces of this sort, generated with Finale, is then imported into Adobe Illustrator. In the case of *This Will Be Changed and Made Solid II*, every musical object that does not mark time in some fashion is generated in Illustrator. The information that is placed on the score in Illustrator is no longer stored in the musical model generated by Finale. The repercussion of this is that the ability to generate parts from a centralized model of the music is no longer possible. As *This Will Be Changed and Made Solid II* is for string quartet, parts had to be generated manually. This process is inherently problematic: if you decide to make a change in the score, you must also make this change in at least one part

(multiplying the amount of work needed to implement this change by at least 2). This very wide feedback loop of semantic change to notational implementation of this change makes an actionable criticality of one's music nearly impossible.

3.4 *This Will Be Changed and Made Solid III*

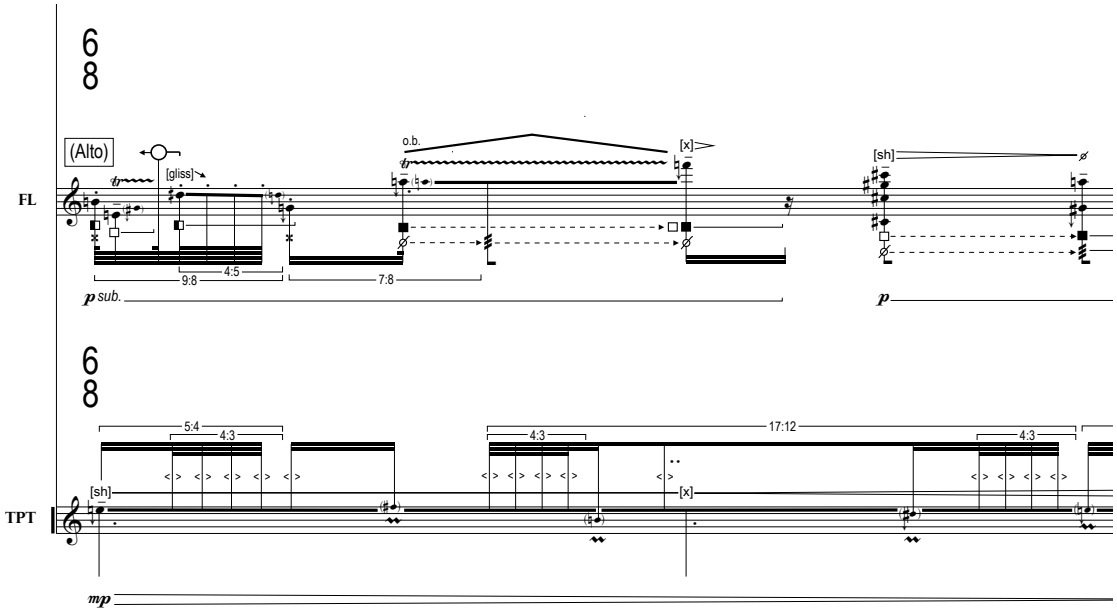

Figure 3.11: Very detailed view: *This Will Be Changed and Made Solid III*.

After producing a sequence of ambitiously notated, semantically unstable, and expensive scores, I was compelled to produce a score with few risks in its reproduction. One reason for this was that the upcoming project, *This Will Be Changed and Made Solid III*, was going to be of a slightly more ambitious musical endeavor, with more performers and a longer duration. While the part preparation of *This Will Be Changed and Made Solid II* was not catastrophic during its preparation, it was clear that the lack of a unified musical model created an unscalable score and part preparation process, particularly as the number of parts increases. In a search for reliability, I generated the entire score in Finale, using the music notational techniques available in the program.

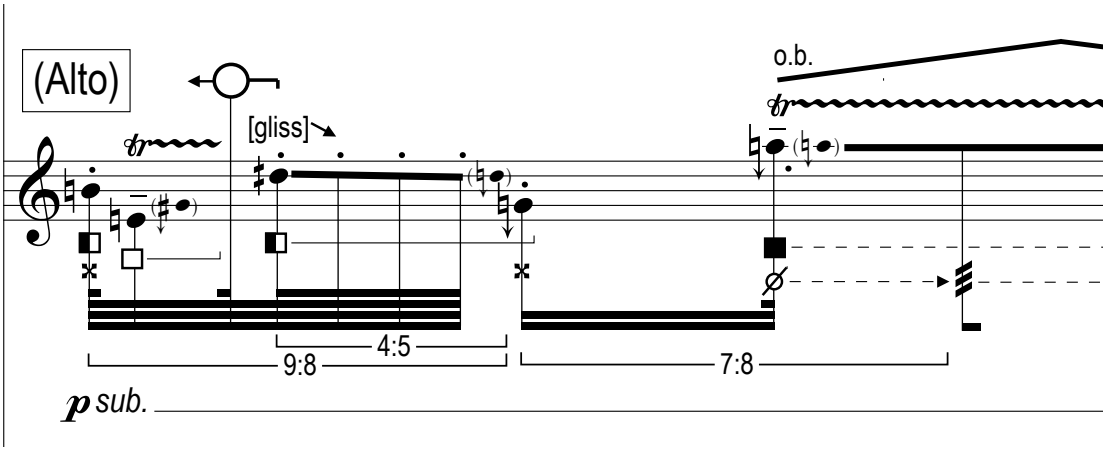

Figure 3.12: Very detailed view: *This Will Be Changed and Made Solid III.*

3.5 *This Is Causing Itself*

While the score and part preparation of *This Will Be Changed and Made Solid III* was made scalable by using a traditional score editor, there were many musical concepts that were not achievable through this system. It became clear that a music notational environment must be able to render more complex graphical environments than in the current music typesetting programs, while still retaining the semantic unification of a rich musical model. Thus began the project that would become **denm**.

Upon completion of *This Will Be Changed and Made Solid III*, I began work on a project that was both a compositional experiment, as well as a fact-finding mission: in search of optimum ways to programmatically generate and organize music notational objects. This project was *This Is Causing Itself*. Fig. 3.13 shows a single panel of a large installation graphical score algorithmically generated in Adobe Illustrator. The score of this work is composed entirely of algorithmically generated grayscale and yellow paths. In the case of this work, there are no traditional music notational symbols, nor a sense of robust organization of these objects. Instead, a low level of understanding the algorithmic control of graphical objects was prioritized.

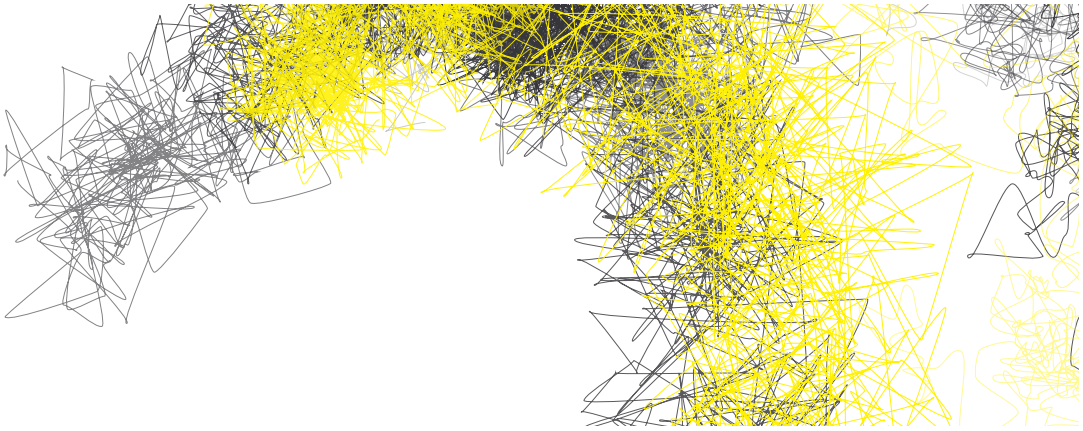

Figure 3.13: Panel view: *This Is Causing Itself.*

3.6 Proof of Concept: *deserving of Songs*

While generating the score for *This Is Causing Itself*, I examined techniques for low-level programmatic generation of graphical objects. The next objective was to generate musical symbols, and then find ways to properly organize them. In Proof of Concept: *deserving of Songs*, the first attempt was made to generate the contents of a full musical score by scripting algorithmically into Adobe Illustrator. This work is written for two flutes, and two contrabasses.

Fig. 3.14 shows a full page of the score of *deserving of Songs*. There were two primary concerns for this process: to create a rich model for woodwind fingering diagrams, with elegantly designed fingering diagram graphics, and to make an automatic translation between fingered-pitch and sounding-pitch staff and graphical tablature representations for string instruments with support for scordatura.

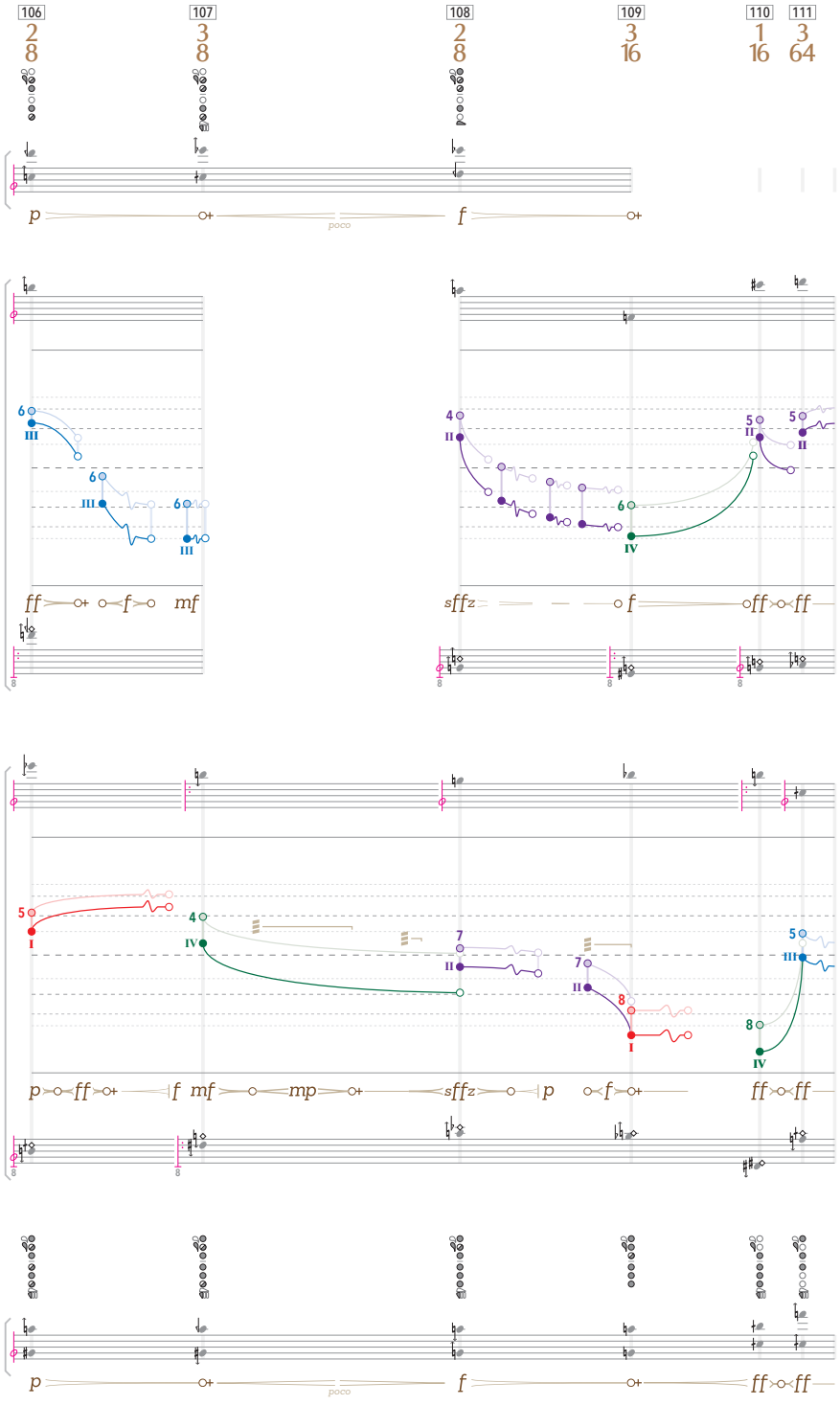

Figure 3.14: Page: Proof of Concept: *deserving of Songs.*

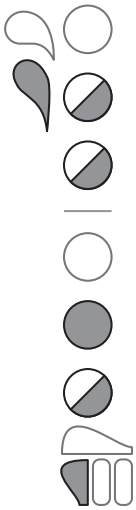

Figure 3.15: Flute fingering diagram: Proof of Concept: *deserving of Songs*.

Fig. 3.15 shows a detailed view of the fingering diagrams used in *deserving of Songs*. The graphic design was inspired by the fingering diagrams generated by Bret Pimentel's Fingering Diagram Builder [9]. In addition to the design and implementation of the graphical representation of the fingering diagram, a robust model of storing key states of the instrument was developed. This model uses JSON [10] to store thousands of monophonic and multiphonic flute fingerings, which where used in this work.

There are several directions that this research will take along with further development of **denm**: for the compositional process, creating an intuitive interface for graphically presenting and choosing strings of fingerings, based on sounding content and the physical actions necessary to enact them, and for the performer, creating an intuitive interface of quickly entering and saving fingering diagrams. Mock-ups of these interfaces were developed in TouchOSC [11], that made the physical activity of entering thousands of fingerings into a database more efficient. A significant amount of interface design strategies for this specific task were learned, though the environment used was inherently unscalable. The lessons learned in this process will be translated into a more scalable environment, for the use with other woodwind families.

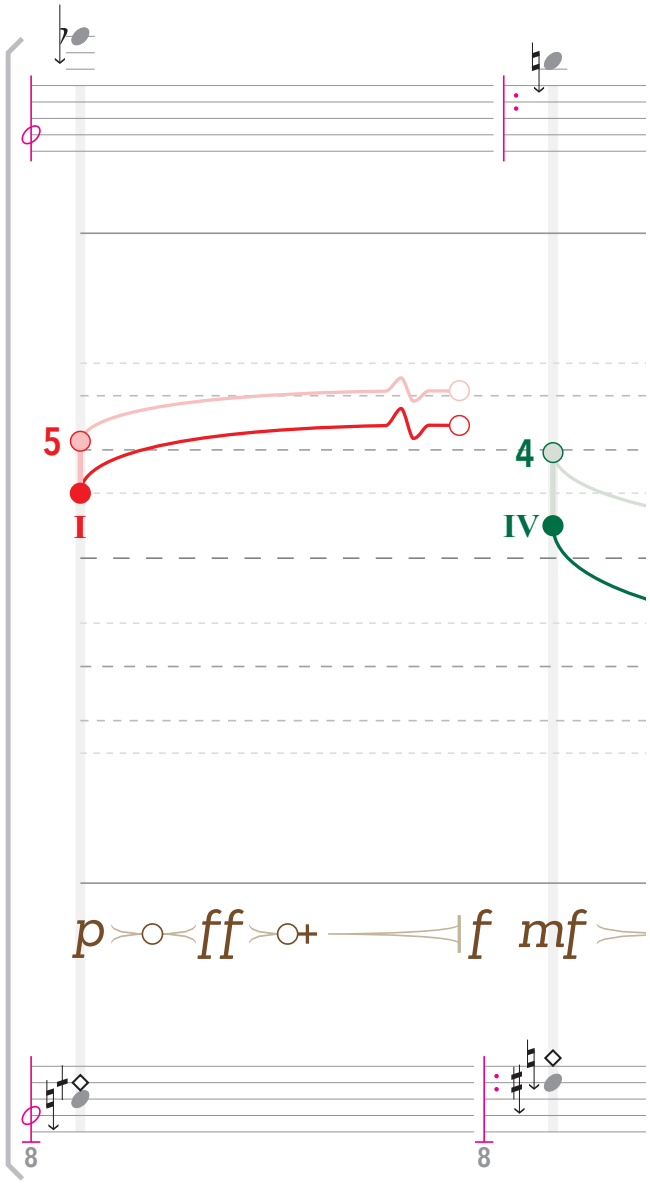

Figure 3.16: Contrabass part: Proof of Concept: *deserving of Songs*.

Fig. 3.16 shows a contrabass part, with three graphs. The topmost graph is the initial sounding pitch of the gesture, accommodating for a scordatura. The middle graph is a string tablature, which shows the finger(s) used, and their placement. The bottommost graph is the initial fingered pitch of the gesture. One technique developed for this application is the automatic selection of clef and transposition optimized for a musical context and a set of instrument-specific preferences.

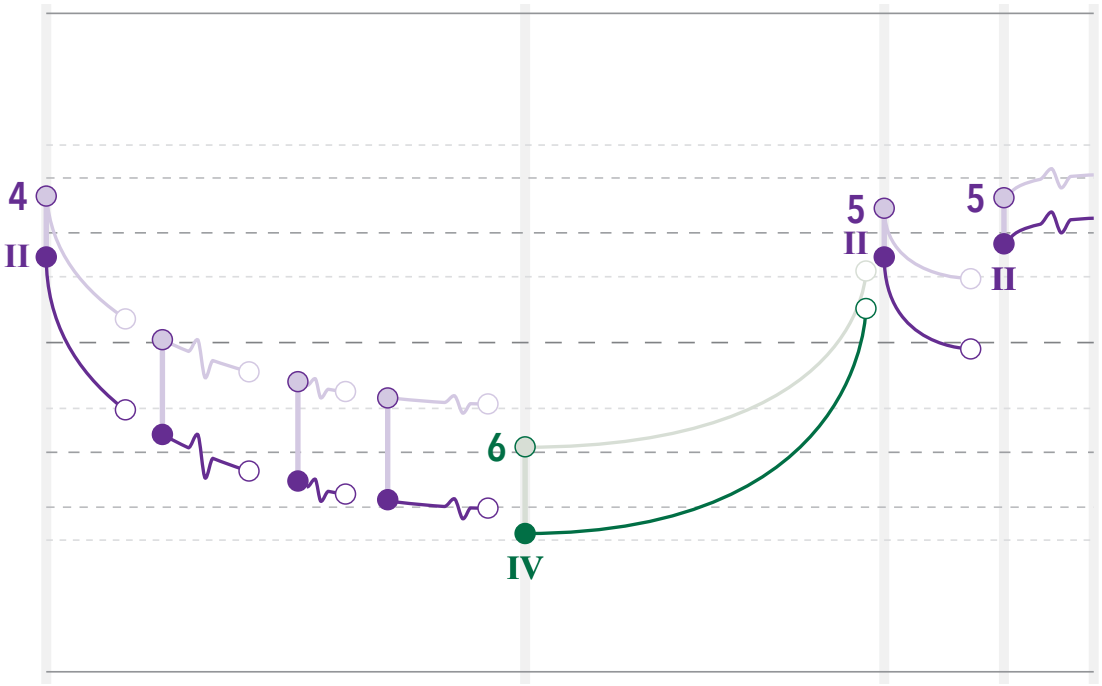

Figure 3.17: Bass tablature graph: Proof of Concept: *deserving of Songs.*

Fig. 3.17 shows the string tablature notation used for the contrabasses. One benefit of string tablature notation is that continuous movement is graphically represented elegantly. Complex physical gestures can be shown with very little translation needed. While relative positions are rendered more effectively with tablature notations, objective placement is often less understandable in this context. In an attempt to solve this issue, a tablature notation system with horizontal dashed lines representing harmonic nodes was developed. In this tablature system, line thickness, dash length, and opacity are each inversely proportional to the partial number that they represent. For example, the most present dashed line that is positioned at the vertical middle represents the second partial (positioned only at $\frac{1}{2} * length$). The least present dashed lines represent the fifth partial (positioned at $[\frac{1}{5}, \frac{2}{5}, \frac{3}{5}, \frac{4}{5}] * length$). This could be extended infinitely, so maximum and minimum partial limits are user-definable.

3.7 Field Test 1: *dry;run.*

Rhythms are conspicuously absent from *deserving of Songs*. Field Test 1: *dry;run.* served as a continuation of development of the previous work algorithmically generating music notation in Adobe Illustrator. A more robust model of rhythm was developed for the score preparation of this work. Beyond the fundamental concepts of rhythm, ensemble organization was highly prioritized in this process. Whereas the layout of *deserving of Songs* was predominately hard-coded with knowledge of the incoming ensemble context, *dry;run.* required a more flexible automated layout system, where systems and pages of a variety of sizes were possible as the amount of performers active in a given measure is dynamic.

In addition to the ensemble organization model developed for this work, the work of a dynamic cue-ing system (discussed in 2.2) is initiated. In the case of this work, a static, version of a variable cue-ing system is implemented. Fig. 3.18 shows selected measures of the flute part. The flute music is at the bottom of the system, with stems facing upward. The other two parts are shown in abbreviated form: only rhythms and dynamics are shown, with stems facing downward.

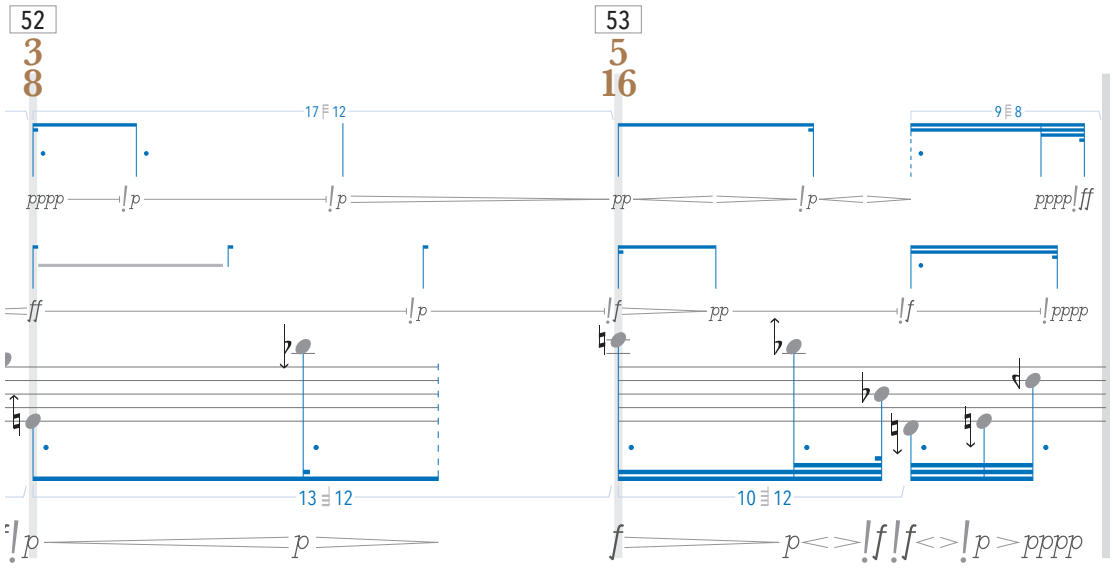

Figure 3.18: Flute part: Field Test 1: *dry;run.*.

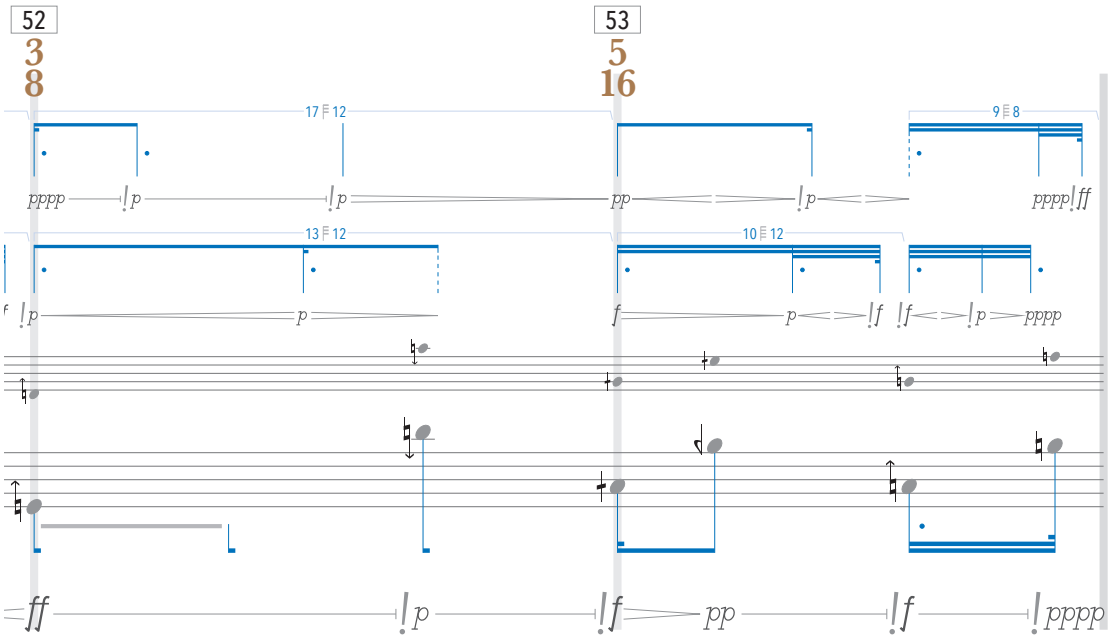

Figure 3.19: Clarinet part: Field Test 1: *dry;run.*.

Fig. 3.19 shows the clarinet part of *dry;run.*. In addition to the abbreviated representation of the other two parts, in the style of the flute part, the clarinetist is also given a secondary staff with a non-transposed (concert-pitch) representation of pitches. In the case of this score, the secondary staff is hardly useful, though it was necessary to investigate the methods of creating an intermediate version of his concept to enable a more scalable implementation in the future. This secondary staff, in tablet-based versions of this software, can be shown and hidden by the performer. It is useful in the case of having a quick reference of a relationship with the pitch content of another player who may not have the same transposition as the B-flat clarinet.

Chapter 4

Graphic Design Strategies

4.1 Depth of Field

4.1.1 Creating Depth of Field

In traditional black-ink-on-paper music notation preparation, it is often challenging to create a functional sense of depth of field of visual elements. As a result, graphic design choices have been made that have negatively impacted a performer's ability to visually parse the music notational image. Because all objects must be black, it is not possible to overlay objects while retaining legibility.

For example, the technique of placing a time signature to the right of a barline catastrophically disrupts the flow of musical time represented horizontally. As shown in Fig. 4.1, the solution of centering the time signature directly above the horizontal point indicating the initialization of a measure avoids disrupting the flow of the horizontal representation of time. This is enabled by changing the design of a barline from a thin black line (with no ability to overlay other objects), to a thick light-gray line, able to be positioned behind the musical information carrying notation objects.

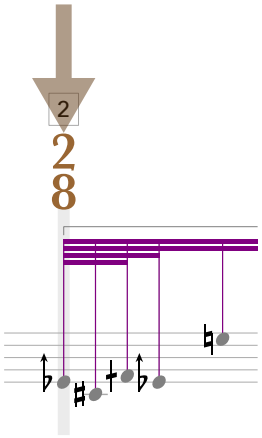

Figure 4.1: TimeSignature centered over beginning of Measure.

There are several strategies used in **denm** that serve to create a functional depth of field:

- Occlusion: the overlapping of objects
- Transparency: the ability to see through objects to others
- Color hue / saturation:
  - Cool and dark colors are perceived in the background
  - Warm and bright colors are perceived in the foreground
- Color contrast: distance on color wheel from other colors in context
- Grayscale value:
  - Light gray values are perceived in the background
  - Dark gray values are perceived in the foreground
- Grayscale contrast: Difference of gray value in relation to objects in context

In Fig. 4.1, one could perceive the depth of field in the order of background to foreground: barline, staff lines, measure number, time signature, stems / beams, noteheads, accidentals. The barline and staff lines are both light gray, putting them

in a position to be perceived in the background. This predisposition is confirmed as multiple objects occlude them (e.g., noteheads, accidentals, stems, beams, etc.). An interesting relationship occurs between the stems and noteheads: while the noteheads occlude the stems, putting the noteheads in a foreground position, the vibrancy of the purple stems gives them emphasis.

The objects that are the furthest towards the background of a notational image should almost disappear to the viewer — their function is not to portray information, but rather create a frame of reference for other objects to portray information (e.g., a staff line is not a representation of a datum in the musical model, but rather it serves as a graphical scaffolding that enables noteheads and accidentals to represent pitch information from the musical model).

4.1.2 Positioning Objects in a Depth of Field

The choice of attributing music notational objects different graphical properties is critical to creating a functional depth of field. Over time, I have found that I tend to use three factors to establish where to place objects in a depth of field: semantic influence, width of scope, and degree of repetition. These choices are generally made intuitively, though they often reveal an implicit formalization in the mapping of abstract musical relationships to perceived graphical space.

Semantic Influence

Some music notational elements have a direct, semantic influence over the elements that occur "after" them (graphically speaking in terms of Common Western Music Notation, to the right of the object). A few examples of the elements that have direct influence on upcoming objects are accidentals and clefs. In these cases, the relationship between the influencing object and influenced object can be seen as destructive (i.e., the modification of the influencing object onto the influenced object must be preserved, or else the material is no longer preserved). If you play a "c-natural" when there is a "sharp" in front of the notehead, you will not be actuating the material properly. If you play the notes on a staff with a treble clef as if it had a bass clef, the pitch material is no longer accurate.

Elements with direct influence over other objects are brought to the foreground of the music notational image. In the case of accidentals, black is used, in a context where noteheads are medium-gray. Clefs, which occur less frequently, are colored red, for maximum contrast with the surrounding environment.

Width of Scope

The scope of a music notational element (and analogously, the scope of the element in the musical model that it is graphically representing) contributes to the speed at which a performer must access the graphical object. For example, information regarding the tempo of a section, with linguistic notes regarding a general affect of a section, may be considered to have a wide scope, and will therefore need to be accessed less quickly than the pitch of an event, which can be considered to have a narrow scope. Types of musical elements that have a wide scope often have an effect on the elements that are contained within this scope, though this influence is often of a non-destructive sense (i.e., you can perform the same musical material at different tempos, and to a certain degree, it will still be acceptable as a representation of the material). Elements with a wide scope (e.g., barlines, staff lines, measure numbers, etc.) may be moved toward the background of the music notational image, as they have no semantic influence over objects in their contained scope.

Degree of Repetition

Graphical objects that have many instances in a single music notational context may be moved further to the background of the music notational image. For example, staff lines are iterative by design. Staff lines have an inherently wide scope, as they comprise the container object of the Staff. As such, staff lines are colored light-gray, to push them to the background and to provide as much of a headroom for informational objects to be placed on top of them as possible. Noteheads of a normal type are not providing information that must be accessed uniquely for each instance. As a result, noteheads are colored medium-gray.

Barlines are also iterative by design, and have wide scope of influence,

though with no direct semantic influence over the objects within its graphical scope. Barlines are merely clarifying markers of information presented in time signatures, and are therefore pushed toward the background of the notational image as possible. Accidentals as a class are highly represented on a staff, though the particular state of an instance of an accidental is highly various, and is critical to defining the graphical objects in its influential scope. Therefore, accidentals are colored black in order to be brought to a more-foreground position than noteheads.

4.2 Dynamic notational elements

While effective graphic design of a music notational image will enhance a performer's relationship to a score, direct manipulation touch interfaces (e.g., iPads) enable the graphical image to be dynamic. As the learning, rehearsing, and performing processes of performers are inherently dynamic, the static image of paper or a PDF is necessarily inhibiting the workflow of a performer. As discussed in Chapter 8, a robust model of laying out musical object is needed for the real-time interactive notational environment for music.

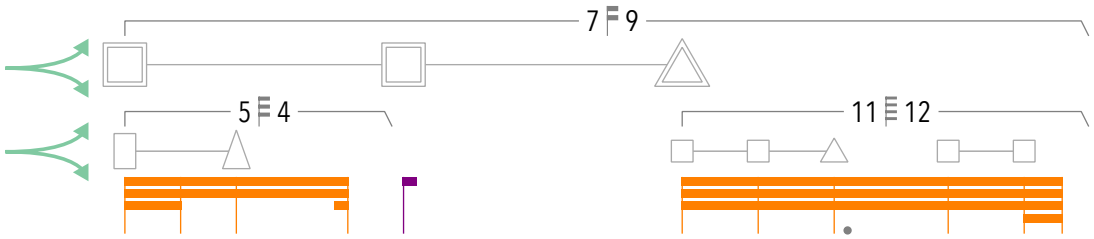

Figure 4.2: Showing of MetronomeGraphicNodes.

An important aspect of a dynamic notational interface is that notational objects can be shown or hidden at any time, and that the layout of the view is updated automatically (and in the best case scenario: elegantly). For example, Fig. 4.2 shows how that when a strand of MetronomeGraphics is shown or hidden, the objects around them are adjusted to use as little space as possible.

Part III

Software Organization and Algorithms

denm seeks to solve inefficiencies in the learning, rehearsing, and performing process of performers that are caused by a static notation of music on a page. While conceptually this is simple, a significant degree of "reinventing the wheel" is necessary to achieve these goals in a truly interactive, intuitive, and elegant way. In this part, we will discuss the current state of Automatic Notation Generators (ANGs), software architecture patterns used, the musical model, the graphical layout model, specific implementations of music notational objects, and the algorithms that analyze the musical for the purposes of alleviating the amount of work normally required from performers to annotate their scores.

Chapter 5

Discussion of Automatic Notation Generators (ANG)

Commercial music typesetting software, such as Finale and Sibelius, are the most common tools for creating musical scores, which require a musician to manually enter musical information via graphical user interfaces. There are cases, for example when compositions are algorithmically generated, where the process of manually entering musical information in this manner is inefficient. As such, programs have been designed to create musical scores where the input from the user is text-based, generated by algorithmic processes, or extracted from spectral analyses.

Most Automatic Notation Generators (ANGs) [12] create a static image, either to be read by musicians from paper, or from a screen displaying it in a Portable Document Format (PDF) representation. LilyPond [13] and GUIDO [14] and convert textual descriptions of music into musical scores. Abjad [15] and FOMUS [16] generate musical score information that can be graphically rendered by LilyPond. OpenMusic [17], Bach [18], PWGL [19] / ENP [20], and JMSL / JSCORE [21] are software tools for composers that generate music notation as part of the compositional process. Belle, Bonne, Sage [22] is a vector graphics library for music notation that enables the drawing of advanced notational concepts not offered by traditional music typesetters. Music21 [23] is a music analysis program that creates score information to be graphically rendered in LilyPond.

Spectmore [12] maps spectral analysis information onto a musical score. A few newer ANGs, such as INScore [24] and LiveScore [25], generate animated musical notation for screen representation.

Thus far, ANGs generate static scores that are useful to composers and theorists, and animated scores that are useful for those performing in real-time (described as the *immanent* screen score paradigm by Hope and Vickory [26]). No ANGs specifically target the rehearsal processes of contemporary music performers (a process described as *interpretive* by Hope and Vickory).

I have found that the most critical period for the success of my own works is the rehearsal processes with performers. Performers spend a considerable amount of time in individual rehearsal and group rehearsal settings, and have developed extensive strategies to comprehend, embody, and execute the propositions of composers (discussed in Chapter 1). Many of the cues that performers notate into their parts are composed of information latent in the musical model—the musical model which is already being represented graphically as the musical score.

Chapter 6

Software Architecture Patterns Used

6.1 Model View Controller

denm implements a software architecture pattern called Model View Controller (MVC). The MVC architectural pattern ensures that information (Model) and its graphical representations (View) are isolated. By decoupling the Model and View elements in a software organization, multiple Views may be generated from the same Model in a flexible and robust manner. A common implementation of this technique is in file management software such as the Finder application on the Macintosh operating system. Files on a computer can be shown either as a list, by icons, or in other ways. When represented as a list, this graphical representation of the information can be sorted by any number of attributes: by name, kind, size, and so on. Regardless of how the information is shown, the data is the same.

The Controller in the MVC pattern is an intermediary layer, situated between the information of the Model and the graphical representation that is View, that offers a user the opportunity to interact with the information. In the example of the Finder application, a user is able to choose which View of the Model they desire (as a list, by icons, etc.). The Controller element sends instructions to the

data source to update its contents and to the View to update its graphical representation of the information. The user interacts only with the Controller element in the MVC architecture, allowing the Model and View to be decoupled entirely, and to protect the data from being manipulated directly by ignorant and/or malicious users.

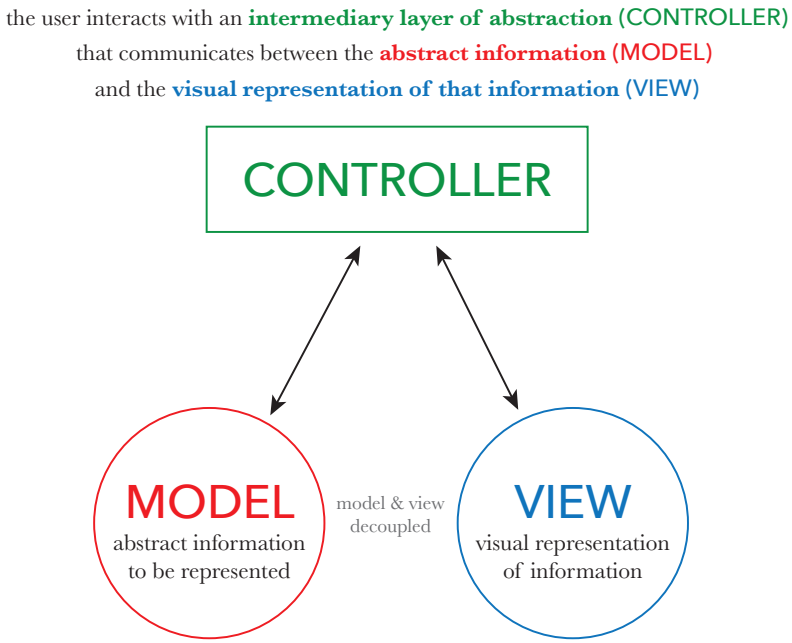

Figure 6.1: Model View Controller (MVC) design pattern.

A similar organization of information and its multiple graphical representations is critical in the preparation of music. A musical score and the parts read by performers are multiple Views of the same Model. The Model of a piece of Common Practice Era Western music may contain abstract data points of pitches, durations, dynamics, articulations, tempo information, and others. The sheet music that is performed from is not a spreadsheet of abstract musical information, but rather graphical representations of it.

For example, pitches are represented graphically with noteheads that are placed on specific points along the y-axis of a staff, conventionally composed of five equidistantly spaced horizontal lines. Further clarification is made with acciden-

tals, objects positioned next to the noteheads that represent adjustments of pitches either up or down in pitch space. Durations are organized with varying amounts of beams and varying graphical styles of noteheads. Dynamics may be graphically represented by a combination of specifically styled ***f***, ***p***, and ***m*** characters, and the interpolations between them may be represented with hairpin graphics, showing relative values of the objective dynamic states.

In traditionally printed Common Western Music Notation, there is simply a Model, with multiple Views (a Score and multiple Parts). As a result of the printing of the View on paper with ink, the graphical representation is static. The performer inherently has no agency in a re-construction of the graphical representation of the musical information.

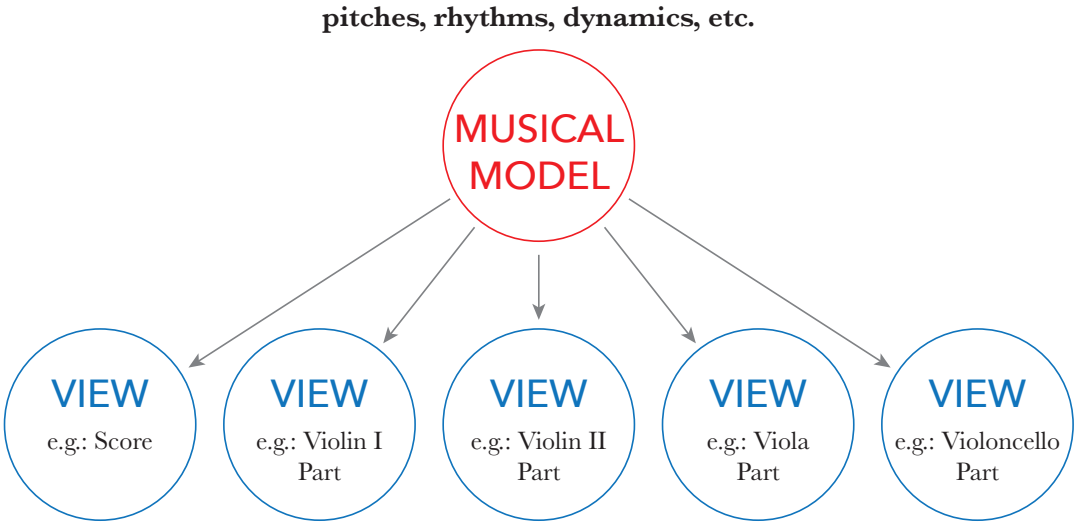

Figure 6.2: Current usage of Model / View separation in music preparation.

Performer's spend a considerable amount of time marking-up their parts in order to provide themselves with a richer graphical representation of the musical environment (see Chapter 1 for further discussion). Because of the inherently static representation of ink-on-paper, the View element of a traditionally prepared score or part is immutable. The information that the performer may be summoning (e.g., abbreviated representations of other players' parts, subdivision markings for complex metrical environments, etc), is not automatically retrievable, and is

therefore re-entered by the performer. There is a cost of time and energy of the re-entering process of secondary notation.

In some cases, each performer in an ensemble context elects to play from a full score, where representations of all parts are visible. This technique may be necessary when the complexity of ensemble interaction is high. For these cases, a performer benefits from the knowledge of what other performers are doing; this knowledge contextualizes their own actions. Having a rich understanding of one's environment aids the intelligent clumping of their part into a higher-order organization of intent. The downside of this technique, however, is that a full-score representation of the environment may be too graphically rich. When this is the case, a cost of access is incurred as the player is forced to parse a complex graphical environment for the relevant information.

The traditional Model / View separation of the musical Model and multiple Views (manifested as a full score and a set of parts) is a simple implementation of the concept. To reach an optimal notational space for the performer that enables a complex set of relationships to an ensemble environment, while graphically representing an essentialized form of these relationships, requires extending the use of the MVC pattern into more subtle realms of the music notational environment.

The ideal context of learning, rehearsal, and performing music via notation synthesizes the richness of environmental information found in a full score (attainable, though, at a cost of parsing its complexity), and the terseness of a single part. **denm** is designed to enable the performer to benefit from a rich model of a musical work, without the cost associated with the previous models of access. By introducing a dynamic View element via a screen score, and a Controller element via a touch-based direct-manipulation interface (e.g., iPad), to the music notational environmental, this synthesis will be possible.

6.2 Tree Structure

Many aspects of the structures in **denm** are modeled hierarchically. A robust and flexible tree model was implemented and thoroughly tested in the Swift

language to enable a scalable architectural design and expressive programming process.

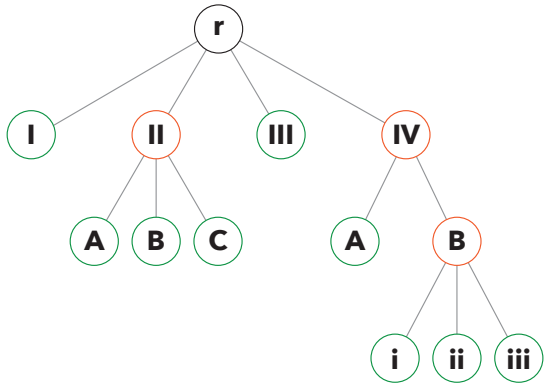

Figure 6.3: Example of tree structure.

Fig. 6.3 shows an N-ary tree structure that is the underlying model of many components in the **denm** software architecture. There are many attributes and operations that are implemented in this model. In the **denm** software, a single class called Node exists that, when linked to other Node instances, can be aggregated into a rich tree structure.

Properties of Node that can be set and retrieved

`parent : Node?`

A `Node` that contains this `Node` instance as a `child`. In Fig. 6.3, the parent `Node` of `r.IV.B` is `r.IV`. All `Nodes` in Fig. 6.3 have a `parent Node` except for the `root Node r`.

`children : Array of Node`

The `children` of a `Node` is a mutable ordered collection of `Nodes`. In Fig. 6.3, the `children` of `r.ii` is `[r.II.A, r.II.B, r.II.C ]`. The `children` array is initialized with a length of 0. `r.IV.A` has 0 `children`, for example.

Properties of Node that are retrievable (computed)

`root : Node?`

The `root Node` of a `Node` has no `parent Node`. If there is only a single `Node` in a tree structure, the single `Node` is its own `root Node`. For every `Node` in Fig. 6.3, the `root Node` is `r`.

`siblingLeft : Node?`

The sibling `Node` to the left of a given `Node` may or may not exist (as notated by the `?`). In the case that the index of the given `Node` in its `parent Node's children` array is > 0, the `Node` with an index of $n - 1$ in the given `Node's parent Node's children` array is returned. Otherwise, the `nil` value is returned.

`siblingRight : Node?`

The sibling `Node` to the right of a given `Node` may or may not exist (as notated by the `?`). In the case that the index of the given `Node` in its `parent Node's children` array is $<$ the amount of `children` contained by the given `Node's parent Node`, the `Node` with an index of $n + 1$ in the given `Node's parent Node's children` array is returned. Otherwise, the `nil` value is returned.

`leafLeft : Node?`

The `leaf Node` to the left of a given `Node` may or may not exist (as notated by the `?`). A `leaf Node`, by definition, has no `children`. In Fig. 6.3, the `leafLeft` of `r.IV.A` is `r.III`, while the `leafLeft` of `r.IV.B.III` is `r.IV.B.II`. The `leafLeft` of `r.I` is `nil`. In this case, the `leafLeft` and `siblingLeft` properties point to the same `Node`.

`leafRight : Node?`

The `leaf Node` to the right of a given `Node` may or may not exist (as notated by the `?`). In Fig. 6.3, the `leafRight` of `r.II.C` is `r.III`, while the `leafRight` of `r.II.B` is `r.II.C`. The `leafRight` of `r.IV.B.iii` is `nil`.

`leaves : Array of Node`

The `leaves` of a `Node` is an ordered collection of `Nodes`. In Fig. 6.3, the `leaves` of `r` are: `[r.I, r.II.A, r.II.B, r.II.C, r.III, r.IV.A, r.IV.B.i, r.IV.B.ii, r.IV.B.iii]`. The `leaves` of `r.IV` are: `[r.IV.A, r.IV.B.i, r.IV.B.ii, r.IV.B.iii]`.

`pathToRoot : Array of Node`

The `pathToRoot` of a `Node` is an ordered collection of `Nodes`, providing the hierarchical lineage from the given `Node` to its `root`, with each `parent Node` in in the path. In Fig. 6.3, the `pathToRoot` of `r.II.B` is `[r.II.B, r.II, r]`.

`height : Int`

The `height` of a `Node` is the distance between the given `Node` and its furthest descendant `Node`. In Fig. 6.3, the `height` of `r.II.B` is 0, while the `height` of `r.IV` is 2.

`heightOfTree : Int`

The `height` of the `tree` of a given `Node` is the `height` property of a `Node's root`. In Fig. 6.3, the `heightOfTree` of all `Nodes` is 3.

`depth : Int`

The `depth` of a `Node` is the amount of `Nodes` in its `pathToRoot` -1.

`isRoot : Bool`

If a given `Node` is the `root` of a `tree` structure (if the parent of the given node is not `nil`).

`isLeaf : Bool`

If a given `Node` is a `leaf` in a `tree` structure (if the amount of `children` of the given `Node` is 0).

`isContainer : Bool`

If a given `Node` has an amount of `children`> 0.

Methods of Node

`addChild (Node)`

Add a `Node` to the `children` array of a given `Node`. This method appends the `Node` submitted through the method call to the end of the given `Node's children` array. The `parent` property of the `Node` submitted through the method call is set to a pointer to the given `Node`.

`insertChild (Node) atIndex (Int)`

Insert a `Node` to the `children` array of the given `Node` at a specified index (if possible). The `parent` property of the `Node` submitted through the method call is set to a pointer to the given `Node`.

`removeChild (Node)`

Removes a specified `Node` from the `children` array of the given `Node`.

`copy -> Node`

Generates a deep copy of the given `Node`. The object returned by this method is identical to the given `Node`, though it is duplicated in memory (the pointers contained within the given `Node` do not point to the same spaces as the pointers in the returned object of this method). This method is to be overridden by each subclass to be tailored to its specific stored data.

`getAncestorAtDistance : (Int) -> Node?`

Returns a `Node`, if present, that is the specified distance up from the given `Node`.

`getDescendantAtDistance : (Int) -> Node?`

Returns a `Node`, if present, that is the specified distance down from the given `Node`.

Chapter 7

Musical Model

The musical model **denm** is designed to be thorough in its representation of Common Western Music Notation concepts, but also extended to support musical concepts implemented by composers of contemporary music. The current implementation of the musical model of **denm** contains `DurationNodes` (described in Sec. 7.4) and Measures. These two elements, unlike in the musical models of some traditional music typesetting softwares, are isolated. DurationNodes contain Components, which are the lowest-level element in the musical model.

7.1 Component

Musical information such as pitch, dynamic markings, articulations, etc., is organized internally as an `enumeration` (`enum`) in the Swift language called `Component`. An `enum` in the Swift language allows related values to be handled in a type-safe manner.

Component Cases

```
case SlurStart(pID: String, iID: String)
case SlurStop(pID: String, iID: String)
case Rest(pID: String, iID: String)
case Dynamic(pID: String, iID: String, marking:  String)
```

```
case Articulation(pID: String, iID: String, marking:  [String])
case Pitch(pID: String, iID: String, midi:  [Float])
```

For example, the Component `enum` contains values like `.Pitch`, `.Dynamic`, `.Articulation`, and others. Each `enum` value may also carry with it specific parameters that are applicable to that particular `enum` value. Every Component value is has `PerformerID` (`pID`) and `InstrumentID` (`iID`) values, as well as any other value-specific information (e.g., in the case of `.Pitch`, a `MIDI` value is passed in the form of a `Float`, or in the case of `.Dynamic`, as dynamic marking value is passed in the form of a `String` (e.g., ``‘‘mf’’`` or ``‘‘ppp’’``), etc.). Some `Component` `enum` values require no extra parameters to be passed, such as the `.SlurStart` component.

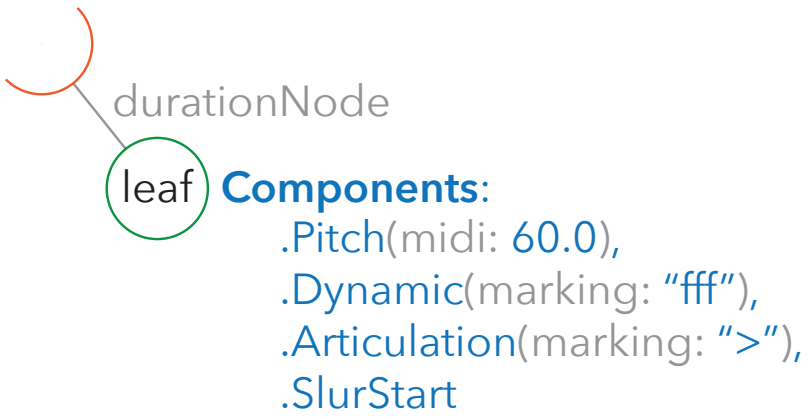

Figure 7.1: Example of Components held by a DurationNode.

The implementation of the `Component` will be extended to support additional types of musical information as these additional types are supported throughout all stages of the softwares architecture (e.g., input, model, and graphical representation).

7.2 Pitch

A `Pitch` in the **denm** musical model can be initialized with either a `MIDI` or `Frequency` value.

MIDI

`MIDI` is a `struct`, which can be converted to and from a `Frequency`.

Frequency

`Frequency` is a `struct`, which can be converted to and from a `MIDI`.

PitchClass

`PitchClass` is a subclass of `Pitch` generated from the `modulo 12` representation of a given `Pitch`.

PitchSpelling

A `Pitch` may or may not be spelled. A `PitchSpelling` contains values `Coarse`, and `Fine`, as well as a `LetterName`. For example, a F quarter-sharp down an eighth tone (one possible `PitchSpelling` of the `MIDI` value of 65.25), has a `Coarse` value of 0.5, a `Fine` value of -0.25, and a `LetterName` of `.F`.

PitchDyad

A collection of two `Pitches`.

PitchInterval

A `PitchInterval` is a subclass of `Pitch` that is created with the difference of the two `Pitches` in a `PitchDyad`.

PitchVerticality

A `PitchVerticality` is a collection of $1 - n$ `Pitches`. Contains ${}_nC_2$ or $\frac{n!}{2!\,(n-2)!}$ `PitchDyads`.

PitchPolyphony

A `PitchPolyphony` is a collection of $1 - n$ `PitchVerticalities`.

7.3 Duration

The concept of `Duration` critical to any musical model. The model of Duration in the current implementation of **denm** is only metrical. That is, all `Durations` have a property of `Beats` and `Subdivision`, which provide a unit of musical time that is relative, converted into an objective sense of time with a `Tempo` (beats per minute) and objective sense of graphical width with a `BeatWidth` (how many graphical points per beat) property. Future implementations of **denm** will include a non-metrical definition of `Duration`.

Comprising a metrical Duration are two properties: `Beats` and `Subdivision`. `Beats` are integer values that are counted by a performer, and a `Subdivision` is an integer value (conforming to the statement 2^n) that dictates the speed at which you count Beats in a given Tempo context.

7.3.1 Beats

Properties of Beats that can be set and retrieved

```
amount :  Int
```

Operator Overloading for Beats

== Check if Beats A is equivalent to Beats B

>= Check if Beats A is greater than or equal to Beats B

<= Check if Beats A is less than or equal to Beats B

> Check if Beats A is greater than Beats B

`<` Check if Beats A is less than Beats B

`+` Returns sum of Beats B to Beats A

`-` Returns difference of Beats B from Beats A

`/` Returns dividend of Beats A by Beats B

`*` Returns product of Beats A by Beats B

`+=` Add Beats B to Beats A

`-=` Subtract Beats B from Beats A

`/=` Divide Beats A by Beats B

`*=` Multiply Beats A by Beats B

7.3.2 Subdivision

Properties of Subdivision that can be set and retrieved

`value : Int`

The value of `Subdivision` in terms of 2^n (e.g., 8, 16, 32, 64, etc).

`level : Int`

The amount of beams in the graphical representation of a `Subdivision` (e.g., if `value` = 8, `level` = 1, if `value` = 32, `level` = 3, etc)

Operator Overloading for Subdivision

`==` Check if Subdivision A is equivalent to Subdivision B

`>=` Check if Subdivision A is greater than or equal to Subdivision B

`<=` Check if Subdivision A is less than or equal to Subdivision B

`>` Check if Subdivision A is greater than Subdivision B

`<` Check if Subdivision A is less than Subdivision B

`+` Returns sum of Subdivision B to Subdivision A

`-` Returns difference of Subdivision B from Subdivision A

`/` Returns dividend of Subdivision A by Subdivision B

`*` Returns product of Subdivision A by Subdivision B

`+=` Add Subdivision B to Subdivision A

`-=` Subtract Subdivision B from Subdivision A

`/=` Divide Subdivision A by Subdivision B

`*=` Multiply Subdivision A by Subdivision B

Properties of Duration that may be set and retrieved

`beats : Beats`

Beats in the Duration

`subdivision : Subdivision`

Subdivision of the Duration

`subdivisionValue : Int`

Amount of beams necessary for graphically representing the Duration

`scale : Float`

The scale of a Duration is initialized to a value of 1.0. It is only modified in the case of embedded tuplet rhythms, where the Durations of the children of a DurationNode have their scale properties set to the sum of the children / the amount of beats in the parent DurationNode's duration. This property is set recursively, allowing children DurationNode's to inherit their parent DurationNode's scale value.

7.3.3 Duration

Methods of Duration

`respellAccordingToBeats (Beats)`

Adjusts the Beats value to the value submitted. The Subdivision value is then adjusted to maintain an equivalent objective duration. For example, if a duration with a Beats amount of 5 and Subdivision value of 16 is respelled according to a Beats amount of 20, the duration then has a Beats amount of 20 and a Subdivision value of 64. In a sense, this a non-destructive operation on Duration, where the

objective metrical duration is maintained, only the representation is adjusted.

`respellAccordingToSubdivision (Subdivision)`
Adjusts the Subdivision value to the value submitted. The Beats value is then adjusted to maintain an equivalent objective duration. For example, if a duration with a Beats amount of 3 and a Subdivision value of 16 is respelled according to a Subdivision value of 64, the duration then has a Beats amount 12 and a Subdivision value of 64.

`getGraphicalWidth (BeatWidth)`
Returns the width of the Duration in points for a screen representation. The width in points of an eighth-note is submitted. A planned extension of this method will take in a tempo function and a BeatWidth, to accommodate scaling of the graphical width of the Duration based on a fluctuating tempo.

`getDurationInMilliseconds (Tempo)`
Returns the amount of milliseconds of a Duration given a Tempo. A planned extension of this method will take in a Tempo Function, which will accommodate scaling of the objective duration by a fluctuating tempo.

`getDurationInSamples (Tempo, SamplingRate)`
Returns the amount of samples of a Duration. This is dependent upon a given Tempo and SamplingRate (how many samples taken per second). This value is used for the internal timing mechanisms for auditory and visual playback.

Operator Overloading for Duration

== Check if Duration A is equivalent to Duration B
>= Check if Duration A is greater than or equal to Duration B
<= Check if Duration A is less than or equal to Duration B
> Check if Duration A is greater than Duration B
< Check if Duration A is less than Duration B

`+` Returns sum of Duration B to Duration A

`-` Returns difference of Duration B from Duration A

`/` Returns dividend of Duration A by Duration B

`*` Returns product of Duration A by Duration B

`+=` Add Duration B to Duration A

`-=` Subtract Duration B from Duration A

`/=` Divide Duration A by Duration B

`*=` Multiply Duration A by Duration B

7.4 DurationNode

Rhythm in **denm** is modeled hierarchically. The `DurationNode` is a subclass of `Node` (described in Sec. 6.2), and therefore inherits all methods defined in the implementation of `Node`. DurationNode is the only class in this tree structure, though it can be a `root`, `container` or `leaf DurationNode`. Events in a musical context are equivalent to `leaf DurationNodes` in this tree structure, while a `BeamGroup` (described Sec. 9.3) is the equivalent to the `root DurationNode`. `Container DurationNodes` act only as abstract organizational elements in this model, which have no direct manifestation as musical material. Instead, container nodes are represented with tuplet brackets, if applicable.

Properties of DurationNode that are settable and retrievable

`id : String?`

A string identifying a DurationNode (e.g., "Violin I")

`duration : Duration`

The metrical `Duration` of a `DurationNode`.

`components: Array of Component`

Collection of `Components` (musical data: pitch information, dynamic marking, articulation marking, etc.)

`offsetDuration : Duration`

Distance in `Duration` from the beginning of the piece. This is an initial implementation of `offsetDuration`. In order to support musics containing alternating metered and unmetered sections, there will be need to be an infrastructure of `Sections`, subclasses of which may be `metered` or `unmetered`. In this case, `offsetDuration` will be the distance in `Duration` from the beginning of the current `Section`, while an additional property of `Section` will be added to `DurationNode`.

`relativeDurationsOfChildren : Array of Int`

An ordered collection of integer values of Beats of the Durations of children DurationNodes.

Class Methods of DurationNode

```
durationNodeRangeFromDurationNodes (Array of DurationNode)
      afterDuration (Duration)
      untilDuration (Duration)
 -> Array of DurationNode
```

Returns an ordered collection of DurationNodes that fit within the specified range of Durations. A common use for this method is to split an entire piece's worth of DurationNodes into only those to be represented in a single System.

`reduceDurationsOfSequence (Array of DurationNode)`

Reduces the Durations of a collection of DurationNodes by the greatest common denominator of the Beats amount. This method first calls to levelDurationsOfSequence: to ensure that the Subdivision values are equivalent of all DurationNodes in the sequence before comparing Beats amounts.

`levelDurationsOfSequence (Array of DurationNode)` Ensures that the Subdivision values of each DurationNode in a collection are equivalent.

`matchDurationsOfSequence (Array of DurationNode)`
This is a public interface that calls to level and reduce the Duration of a collection of DurationNodes.

Initialize a DurationNode

`init(duration: Duration)`
Create a DurationNode with a Duration.

`init(duration: Duration, sequence: NSArray)`
Create a `DurationNode` with a `Duration`, and an arbitrarily deep nested collection of Integers, or Array of Integers. The sequence in this case is similar to the `rtm` model of OpenMusic [17].

A simple example of this is:
`init(duration: Duration(3,8), sequence: [4,2,1])`

Whereas an example of a embedded tuplet is:

```
init(
      duration: Duration(3,8),
      sequence: [[4,[2,1,[2,[1,2]]]],[2,[1,1,1]],1]
)
```

Methods of DurationNode

`addComponent (Component)`
Adds a musical component to the given `DurationNode` (pitch information, dynamic marking, etc.)

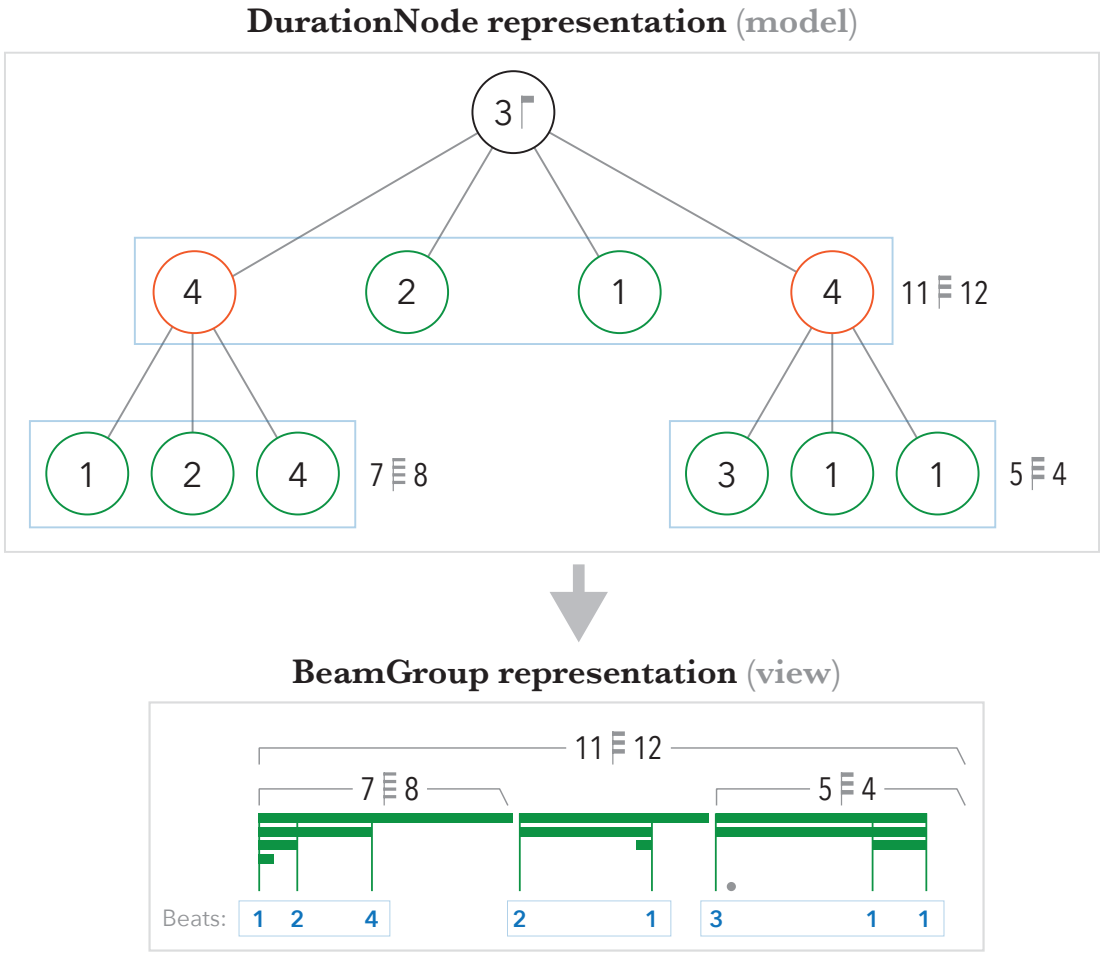

Figure 7.2: Example of DurationNode and corresponding BeamGroup.

In Fig. 7.2, a representation of a complete tree structure model of a rhythm is presented above a score representation of the rhythm. In the next figures, we will examine each `container DurationNode` in isolation, in order to demonstrate a few of the processes that are required to build a graphical representation of arbitrarily complex nested-tuplet rhythms.

In the model representation of these `DurationNodes`, `container DurationNodes` are displayed with orange circles, whereas `leaf DurationNodes` are displayed with green circles. Likewise, in the score representation of these `DurationNodes`, `container DurationNodes` are displayed with orange beams (only the case in Fig. 7.4), and `leaf DurationNodes` are displayed with green

beams.

The `DurationNodes` comprising each level of depth in the model representation of this `DurationNode` tree structure are outlined with a pale blue rectangle. Only the relative durational values (e.g., `[4,2,1,4]` for level of `depth = 1`) are shown in the `DurationNodes` that are not the `root DurationNode`. Similarly, the relative durational values of the `leaf DurationNodes` are displayed, also contained within pale blue rectangles (e.g., `[1,2,4],[2,1],[3,1,1]]`). The `container DurationNodes` are not represented with orange beams in the score representation, as the `container DurationNodes` are represented with tuplet brackets (e.g., `11:12, 7:8, 5:4`).

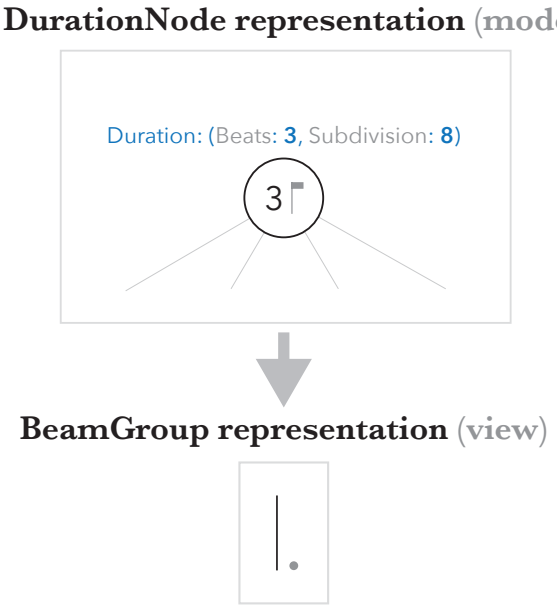

Figure 7.3: Root DurationNode in 7.2.

In Fig. 7.3, a model and score representation of the root `DurationNode` of this rhythm are shown. This `DurationNode` carries with it a `Duration` of `(3,8)`, or three beats with a subdivision of eighth notes. If this `DurationNode` were to be represented in a score, a dotted quarter note would be displayed. As we will see as `DurationNodes` are added to the `root DurationNode` as `children`, the basic `Duration` of `(3,8)` will be modified to accommodate the necessities of the newly added `DurationNodes`.

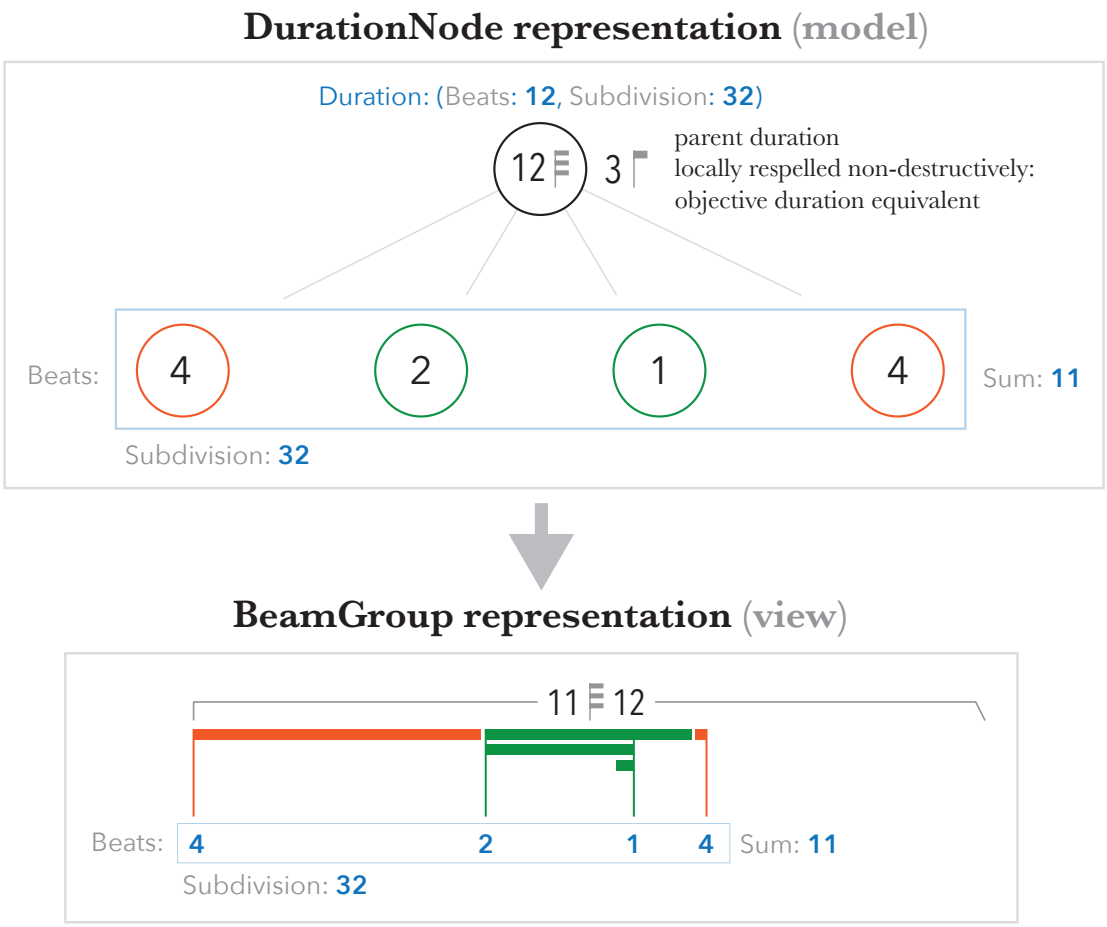

Figure 7.4: Children of root DurationNode in 7.2.

In Fig. 7.4, the beams of `container DurationNodes` in the score representation of this rhythm are colored orange. This type of representation will never be displayed as the actual musical material, as the container `DurationNodes` do not serve as events to be actuated by the performer, but rather they function as abstract organizational elements. This type of representation will be displayed, however, in a user-defined context wherein the underlying structures of an embedded rhythmic context may be useful in the learning process. As `DurationNodes` are added to a `DurationNode` as `children`, the `Durations` of the `parent DurationNode` as well as the `children DurationNodes` may necessitate adjustment. For example, in Fig. 7.4, the `Duration` of the `root DurationNode` has been changed from `(3,8)` to `(12,32)`.

DurationNodes may be added with Durations in any form (i.e., they may or may not have the same Subdivision as the parent). Further, as more DurationNodes are added, their cumulative values may no longer be compatible with the previous representation of Duration held by the parent.

When DurationNodes are added, the Duration of each child DurationNode is leveled to match the others. That is, in whatever form the Duration of each child DurationNode is when it is added (e.g., with a Subdivision of 8ths, 16ths, or 32nds, etc.), it must be adjusted to match the others. This process is undertaken to ensure that the Durations of a set of children DurationNodes can be compared properly. Once the Duration of each child DurationNode is calibrated to the same Subdivision, the Durations are then reduced. This process entails checking each Duration to see if it is in its most reduced form (e.g., a Duration of (2,32) can be reduced to (1,16), or a Duration of (6,16) can be reduced to (3,8)). If all of the Durations can be reduced, the greatest common denominator of all Durations is found, and each Duration is divided by this greatest common denominator value. For example, an array of children DurationNodes with relative durational values of [2,4,2,6] will be reduced to attain Beats values of [1,2,1,3], with the Subdivision values of each being adjusted to maintain the same objective Duration (e.g., ensuring that a Duration before reduction has a value of (2,64) has a value of (1,32) after reduction).

Once the Duration of each child DurationNode is leveled then reduced, the sum of the Beats values of the Durations is calculated. In the instance of the children of the root DurationNode of this rhythm, as shown in Fig. 7.4, the sum of the Beats values of the Duration is 11. The sum of the Beats values of the Durations of the children DurationNodes is then compared to the Beats value of the Duration of the parent DurationNode. In this case, the sum of relative durational values of the children is 11, while the Beats value of the Duration of the parent DurationNode is 3.

If the Durations of the child DurationNodes were to be scaled to match the current Subdivision of the Duration of the parent DurationNode, the results

would be cumbersome; the first `DurationNode child` would have a `Duration` with a value of `(4,8)`, or a half note. To remedy this problem, the representation of the `Durations` of the `parent` and `children` may require modification.

There is often much debate among performers and composers how to spell certain tuplet groupings in certain ways (e.g., `7:4` or `7:8`). **denm** always chooses the closest match between the sum of relative durational values of the `children` and the `Beats` value of the `parent` (e.g., `7:8`, not `7:4`). The reason for this is, particularly in the case of embedded tuplet rhythms where durational impact of tuplets multiply recursively, that the beam representation of the `Subdivision` values can become wildly misrepresentative of the `Duration` at hand. For example, a `Duration` that is technically shorter than another `Duration` that is immediately adjacent, may have a beam representation indicating that it is multiple times longer than the adjacent `Duration`.

The `Durations` of the `children DurationNodes` must be matched to the `parent DurationNode`, and vice versa. In the case that the the sum of the relative durational values of the `child DurationNodes` is greater than the `Beats` value of the `parent DurationNode`, an array of integers is generated, composed of values of $beats * 2^n$ where $0 < n < 5$, and the closest match in this array is found to the `Beats` value of the `parent DurationNode`. For example, the array created for the `container DurationNode` in Fig. 7.4 is `[3,6,12,24,48,96]`. The closest match here is 12. In the case that the sum of the relative durational values of the `children DurationNodes` is less than the `Beats` value of the `parent DurationNode`, the same process occurs as before, though with the values of the sum of the relative durational values of the `children DurationNodes` and the `Beats` value of the `Duration` of the `parent DurationNode` are switched. In this case, an array of integers is generated, composed of values of $sum * 2^n$ where $0 < n < 5$.

This is a static implementation for this process, and will be made dynamic as development continues. Instead of generating a fixed array with six values, each value to be compared will be generated only as needed. In the case of this `container DurationNode`, the values `[24,48,96]` were unnecessarily added to the array, adding to the computation time of finding the closest match.

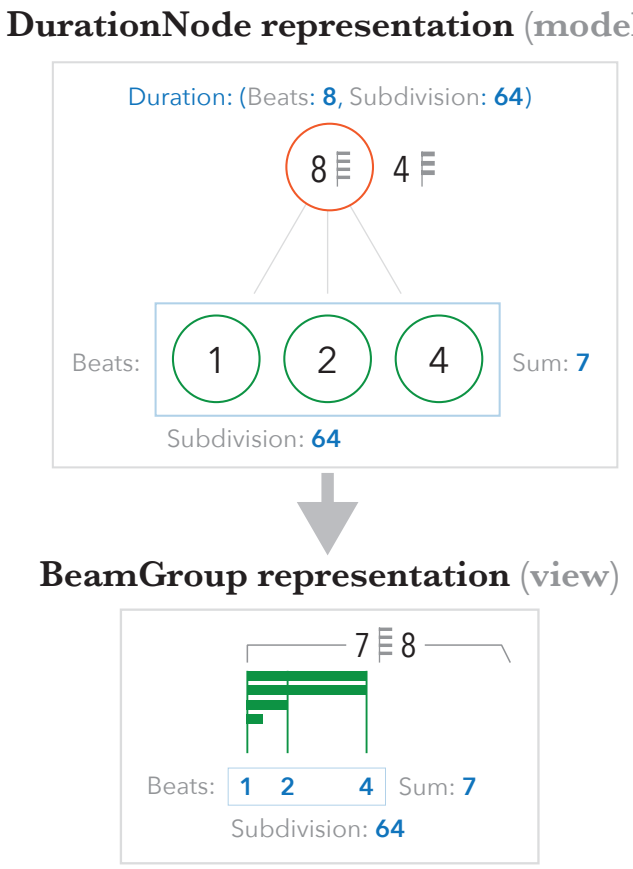

Figure 7.5: Example of DurationNode and corresponding BeamGroup.

Fig. 7.5 demonstrates the addition of `DurationNodes` to an already embedded `DurationNode`. As before, the `Duration` of the `parent DurationNode` and the `Durations` of the `children DurationNodes` have been modified to provide the optimal `Subdivision` relationship, and therefore beaming representation. As can be seen in this example, the choice of `7:8` is made, rather than `7:4`. If `7:4` was chosen, the beaming representation of the `Duration` of the last `DurationNode` in this sequence would indicate a `Duration` roughly twice as long as the the `Duration` of the next `leaf DurationNode` of this `DurationNode` tree structure. As can be seen in the proportionately spaced rhythmic representation of Fig. 7.2, these two adjacent `Durations` are quite similar. If the scaling of `Durations` in a tuplet context is slight, the beaming representation should reflect this.

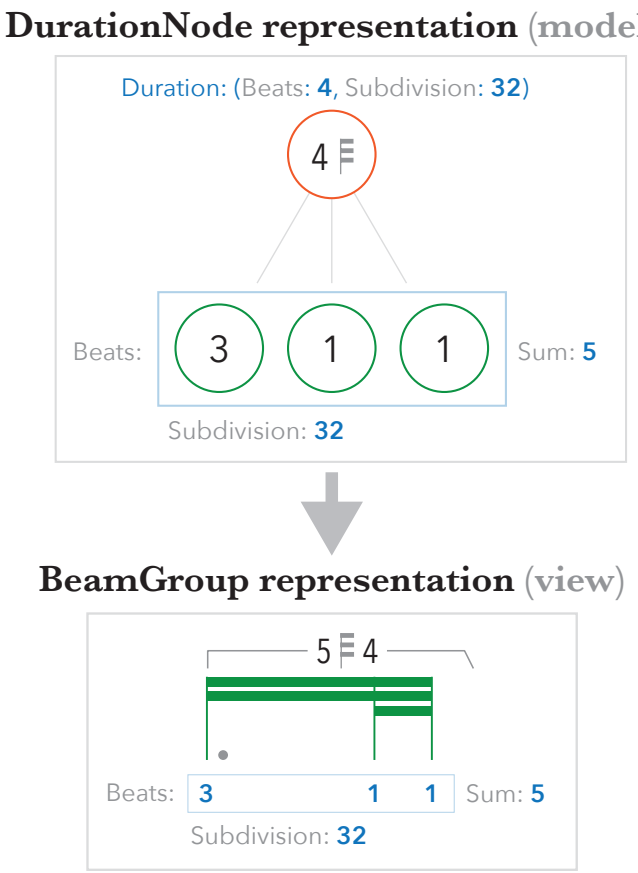

Figure 7.6: Example of DurationNode and corresponding BeamGroup.

In Fig. 7.6, no adjustment of `Durations` is necessary, as the sum of the relative durational values of the `child DurationNodes` is already optimally matched to the `Beats` value of the `parent Duration`.

Chapter 8

Graphical Layout Model

There are three fundamental types of graphical objects in **denm**: ViewNode, Ligature, and Placed. ViewNode objects are organized hierarchically for the purposes of automated layout, Ligature objects connect two ViewNodes, and Placed objects are positioned statically, as they graphically represent a datum in the musical model.

8.1 ViewNode Objects

The ViewNode class implements the methods and properties of the Node class of an implementation of a tree structure described in Sec. 6.2. As such, the ViewNode organization permits hierarchically automated layout of graphical objects. In order to make a score fully interactive, the layout of the graphical objects comprising it must be fully automated. For example, if a performer needs to see a certain type of information at a given time, they must be able to insert it the correct position, without the overlapping of objects. Automated layout will make possible continual optimization of screen real estate and visual clarity of musical objects, regardless of the scope of changes made by the user. ViewNodes are flexible in the way that they lay out their objects, in ways similar to the text-alignment methods of text editors. A parent ViewNode is able to resize according to the children ViewNodes that they contain, as to achieve an optimum screen usage.

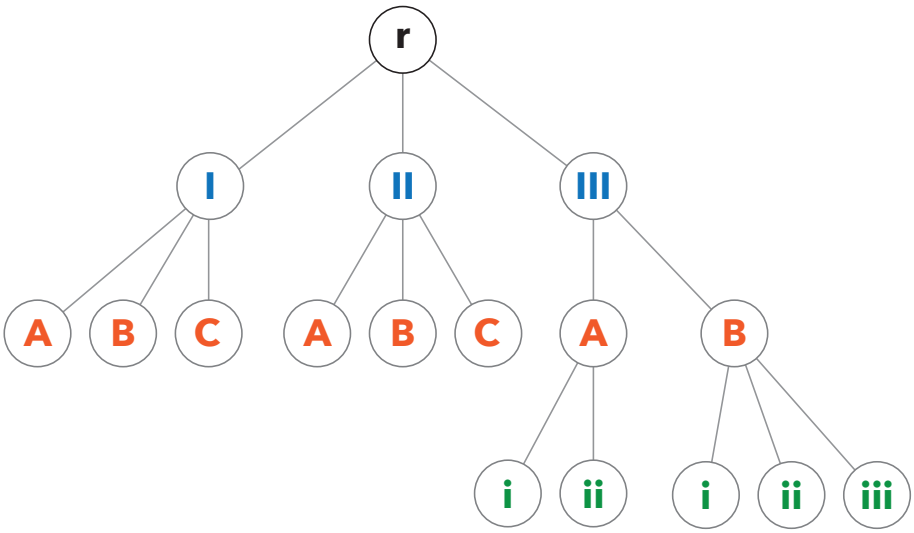

Figure 8.1: Model Representation of ViewNode Tree.

Fig 8.1 shows a model representation of a complete `ViewNode` hierarchy. In the following examples, we will see the graphical repercussions of removing `ViewNodes` from a `ViewNode` tree. When a change is made anywhere within a `ViewNode` tree structure, each `ViewNode` in the `pathToRoot` array of of the modified `ViewNode` recalculates its size and lays out its contents. Each `ViewNode` has several properties that can be set to modify the behavior of the `layout` method. In Fig. 8.2, we see a graphical representation of the `ViewNode` tree model shown in Fig. 8.1.

Dashed lines with arrows in Fig. 8.2 represent the direction of `layoutAccumulation_vertical` or `layoutAccumulation_horizontal` if applicable, while solid lines with arrows represent the direction of `layoutFlow_vertical` or `layoutFlow_horizontal` if applicable. `layoutAccumulation` refers to the direction which the `ViewNode` stacks its `children ViewNodes`. `layoutFlow` refers to the direction which the `ViewNode` aligns its `children ViewNodes`.

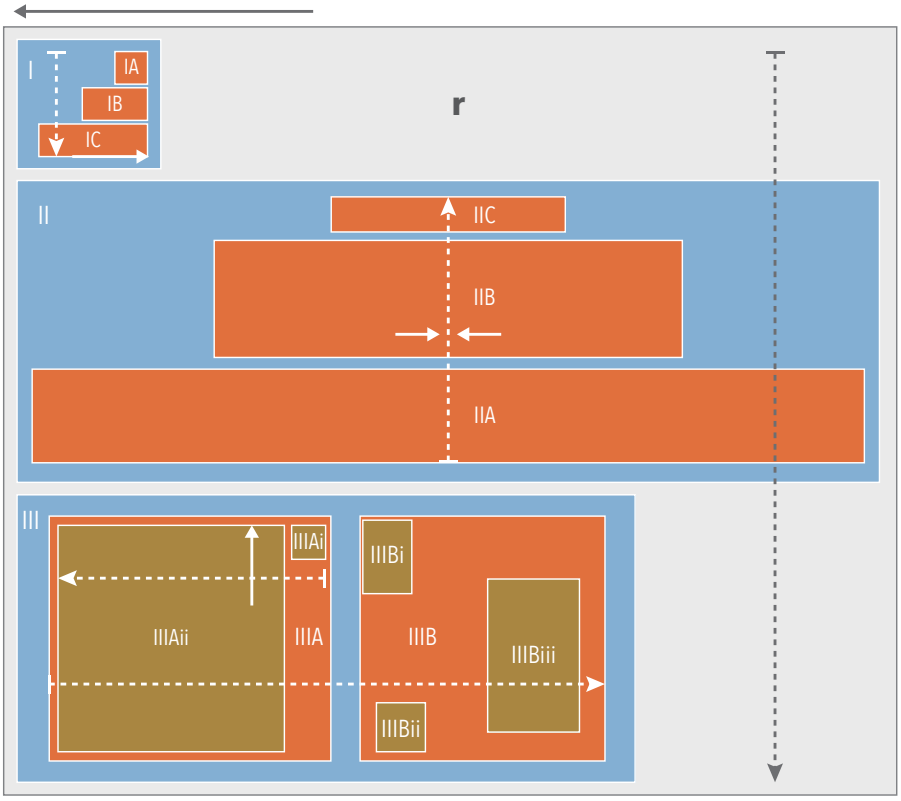

Figure 8.2: View Representation of ViewNode Tree.

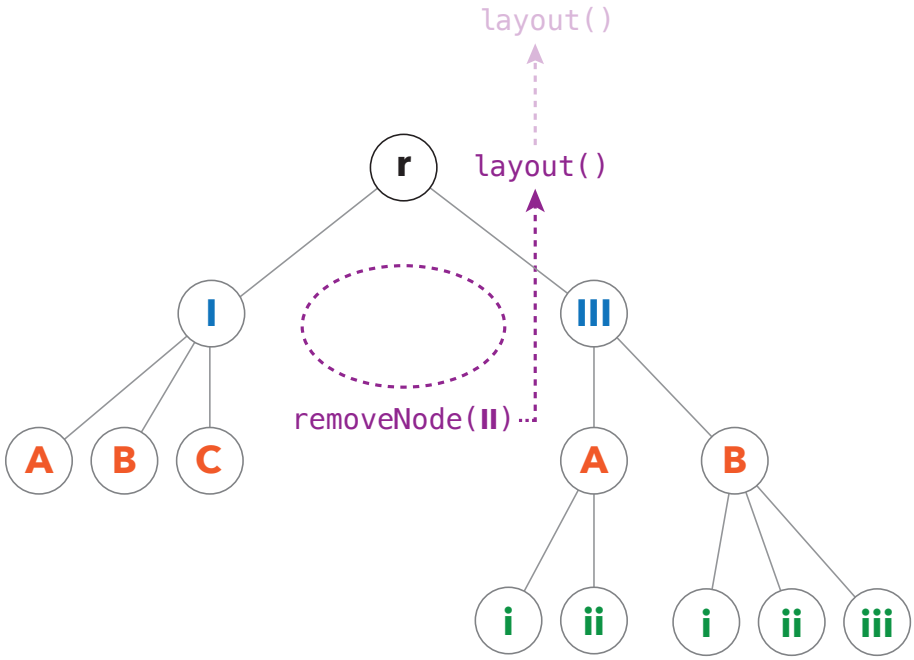

Figure 8.3: ViewNodeTree.

In Fig. 8.3, `r.II` is removed. Fig. 8.4 shows the graphical repercussion of this change in the model. `r.III` is animated upward to fill in the space occupied by `r.II`, and the total size of `r` is adjusted to fit the contents.

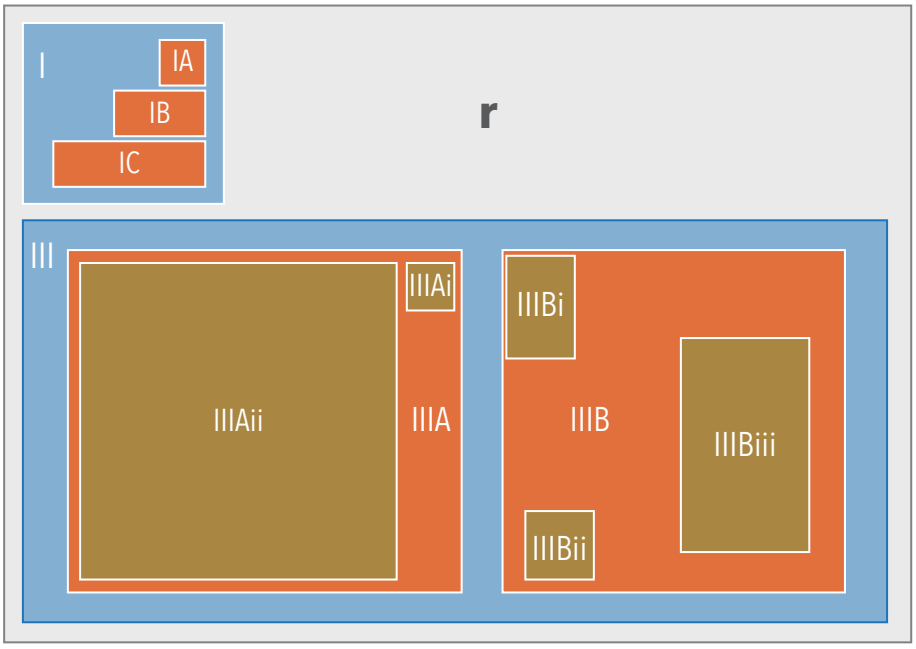

Figure 8.4: ViewNodeTree.

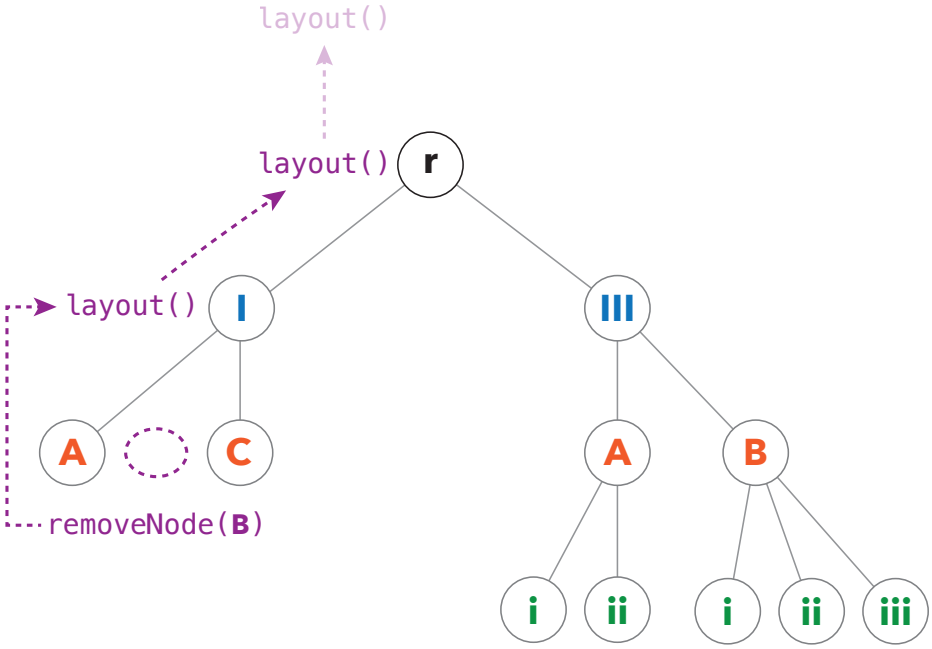

Figure 8.5: ViewNodeTree.

In Fig. 8.5, `r.I.B` is removed. See in Fig. 8.6, that r.I has fit itself to its contents. When a change occurs to a `ViewNode`, a call is made up to its `parent ViewNode` to adjust its size and lay out its contents. These calls are made recursively all the way up to the `root ViewNode` of a `ViewNode` hierarchy.

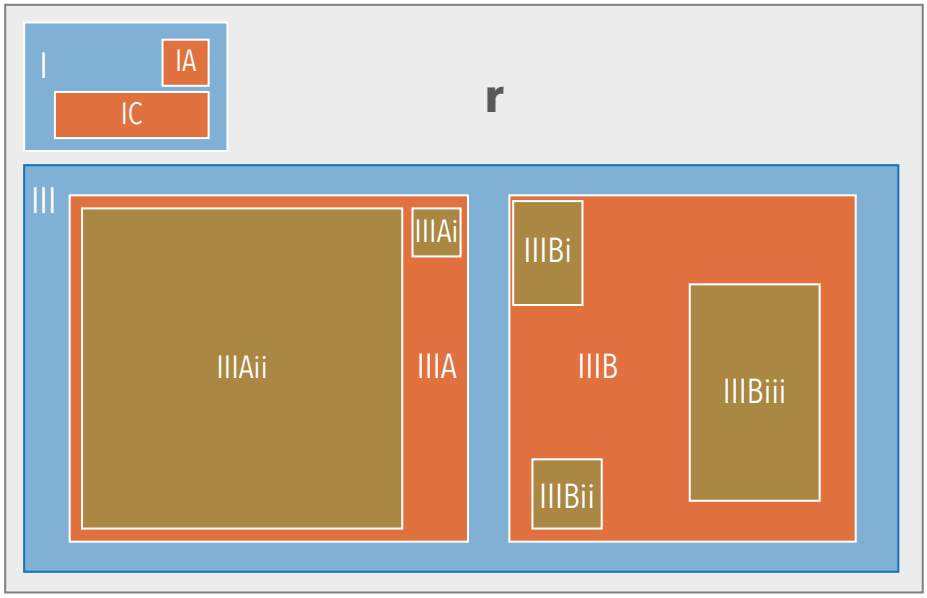

Figure 8.6: ViewNodeTree.

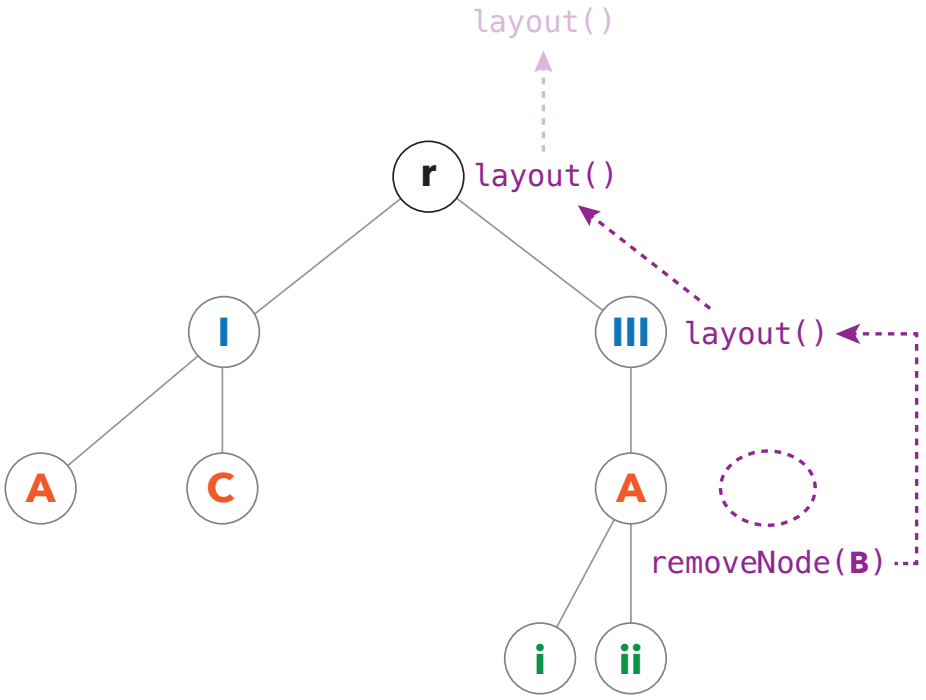

Figure 8.7: ViewNodeTree.

In the previous examples, `ViewNodes` have been removed and the overall height has been adjusted. In Fig. 8.8, `r.II.B` is removed, causing `r.II` to adjust its width. As a result, `r` adjusts its width inward.

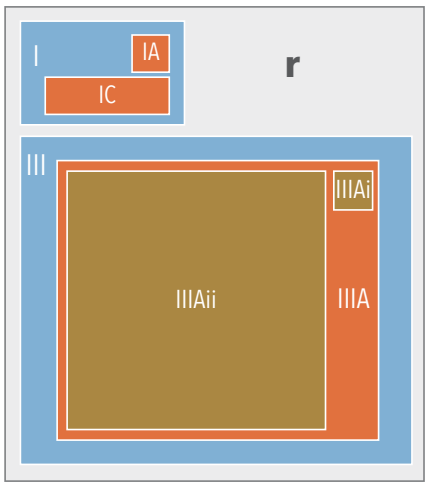

Figure 8.8: ViewNodeTree.

8.2 Ligature Objects

Ligatures connect two ViewNodes. While ViewNodes may be laid out automatically, ligatures are dynamically resized to maintain a connection between these ViewNodes. Ligatures are contained by the parent ViewNode of the two ViewNodes that it connects. One example of a Ligature in the **denm** graphical environment is a Stem. A Stem connects BeamGroup Strata to Graphs. While the distance between a Graph and a BeamGroup Stratum may change due to the insertion of another ViewNode (such as a stratum of Dynamics or other markings), the Stem must be adjusted accordingly. While a ViewNode lays out its content, the path of a Ligature is animated to an adjusted path with the proper coordinates enabling the connection of its two ViewNode references.

8.3 Placed Objects

`Placed` objects are not moved when their superlayer lays out its contents. Most often, `Placed` objects represent data in the musical model. For example, noteheads and accidentals are placed specifically to represent pitch information.

Chapter 9

Music Notational Objects and their Organization

Each music notational object supported thus far by the **denm** musical model has been redesigned specifically for use in an interactive context.

9.1 Structural Organization

Certain objects in a musical score provide a context for other objects to portray musical information, rather that portraying musical information themselves. Objects of this sort are considered `Structural Organization` objects.

9.1.1 PerformerInterfaceView

The primary organizational element of **denm** is the `PerformerInterfaceView`. The `PerformerInterfaceView` is the Controller element in the MVC design pattern (discussed in Sec. 6.1. The `PerformerInterfaceView` manages all of the `Pages` that a viewing performer may interact with. User interface actions are delegated through this class, which modify the View of the musical information with input from a user. For example, a user may turn pages either by touching the screen in designated areas, or connecting a Bluetooth foot-pedal to the tablet device. Further, a user may

initiate the showing or hiding of a specific graphical objects via this controller mechanism.

9.1.2 Page

As Page is a subclass of ViewNode, its size is managed automatically as objects are removed or inserted anywhere within the graphical object organizational hierarchy. The Page is the organizer of a single screen's worth of music. The `Page` contains $0 - n$ Systems.

9.1.3 System

The System is a subclass of ViewNode, and therefore its size is managed automatically as objects are inserted or removed anywhere within its ViewNode hierarchy. A System contains as much music as can be fit horizontally within the bounds of the given screen.

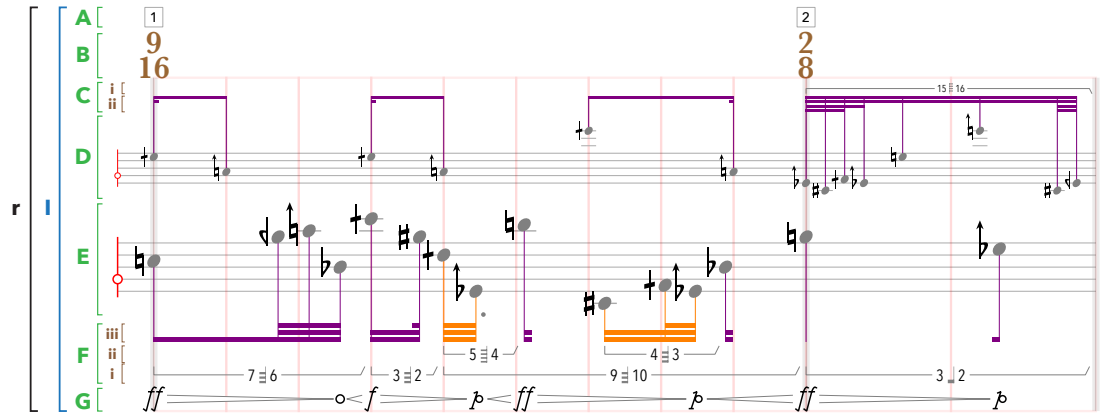

Figure 9.1: Score Representation of System.

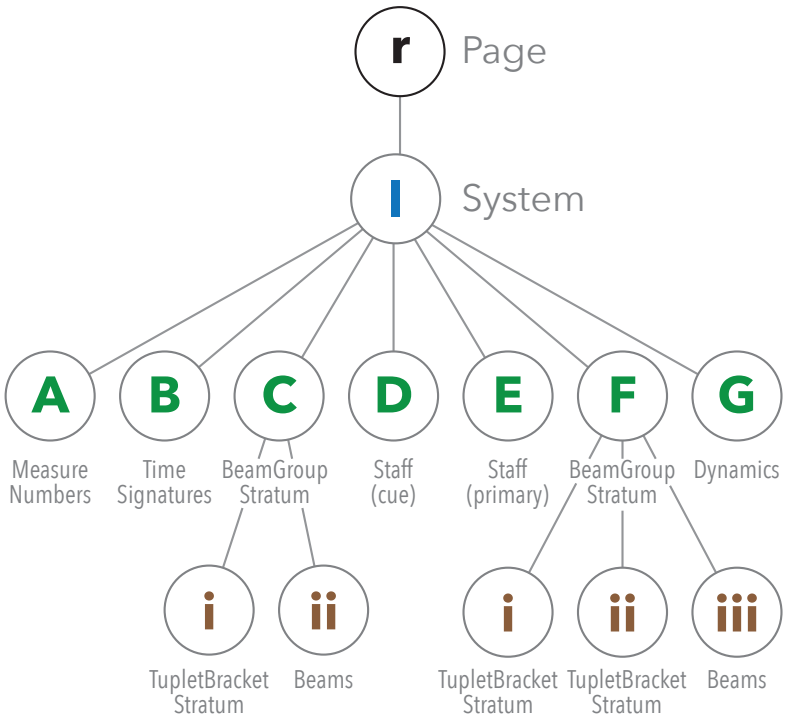

Figure 9.2: Model Representation of System.

9.1.4 Measure

`Measures`, in the **denm** Musical Model (discussed in Chapter 7), as well as in the graphical domain are decoupled from the musical material that it may seem as they contain. Instead, `Measures` are a separate stratum of information layered on top of a layer of `DurationNodes` in the Model, and BeamGroups in the View. The musical models of traditional typesetting software organizes the musical materials as being contained within the hierarchy of measures. This becomes problematic when rhythms may exist outside of the hierarchy of measures, or the beaming of rhythms is to extend across barlines. When using traditional typesetting software, this organizational constraint poses a problem for the user, who will ultimately be required to "break" the organizational constructs of the musical model, creating music notation that will not be scalable (e.g., parts will require readjustment, etc.).

MetricalGrid

A MetricalGrid is a subclass of Ligature, which adjusts its length with an update in the layout of a System. A MetricalGrid shows the beats of Measure. The object consists of an almost transparent line laid over the top of all of the musical information in a measure. The MetricalGrid serves as an immediate indicator of the metrical structure of a measure. Performers often draw lines at the points of beats of a measure to clarify rhythmic proportions of events held within the measure.

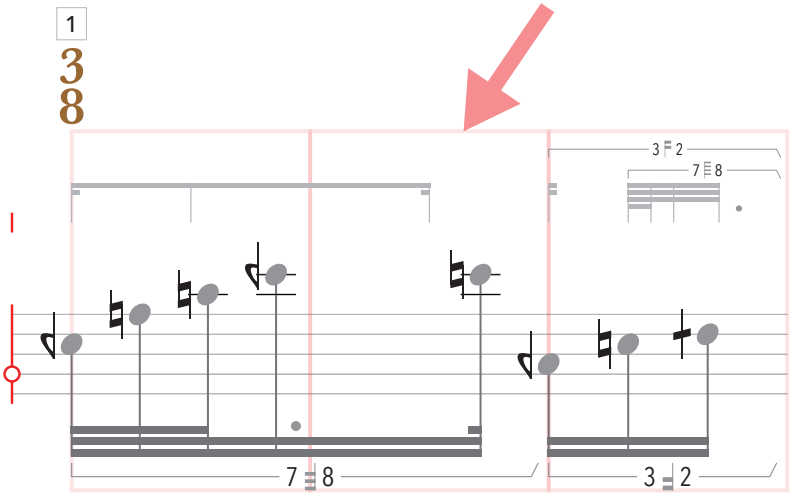

Figure 9.3: Metrical Grid.

MetricalGrids may be engaged for any measure by tapping on the TimeSignature of the given measure. With further development, a user will be able to show or hide all of the MetricalGrids in a given level of structural organization (e.g., `Measure`, `System`, `Page`, or the entire score).

TimeSignature

The TimeSignature in **denm** is centered directly over the starting point of its parent Measure. In traditional music engraving practice, the time signature is placed on a staff, to the right of a barline. This approach consumes a considerable

amount of horizontal space that is coupled tightly (or in the case of **denm**, directly) with the horizontal representation of time. The consequence of this loss of horizontal space is not just a decrease in the amount of material that can be placed on a system, but also an unstable space-to-time relationship of graphical objects and the durations that they represent. By placing the TimeSignature above the musical material, no horizontal space is lost, while a greater isolation of structural information (i.e. meter) and content (e.g., rhythm, pitch, etc.) is achieved. TimeSignatures are organized in a TimeSignatureNode subclass of ViewNode.

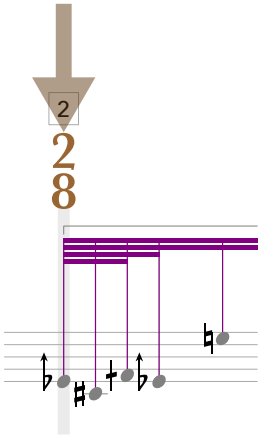

Figure 9.4: TimeSignature centered over beginning of Measure.

Barline

The Barline is a subclass of Ligature, which adjusts its length when the layout of its parent System is modified. The Barline serves to mark the beginning and ending of a Measure. In **denm**, the Barline is designed such that it is perceived in the background of the total notational image. As the a Measure is considered Structural in the graphical model of **denm**, its elements are designed to be perceived as environmental, rather than informational. In **denm**, there is no horizontal space given to the TimeSignature, as is the case in traditional music engraving practice. For further discussion of the reasons behind this choice, see Sec. 4.

The musical information that starts at the onset of a Measure starts at ex-

actly the left point of the Measure, therefore it is placed on top of the Barline. In traditional typesetting, this would cause a collision with multiple black-ink components that would make legibility poor. As such, the Barline in **denm** functions as a subtle confirmation of the structural properties of a metrical structure (i.e. a Measure), without competing visually with the information. The color of the Barline is very low in contrast to the background color environment, and very high in contrast to the informational elements layered above it.

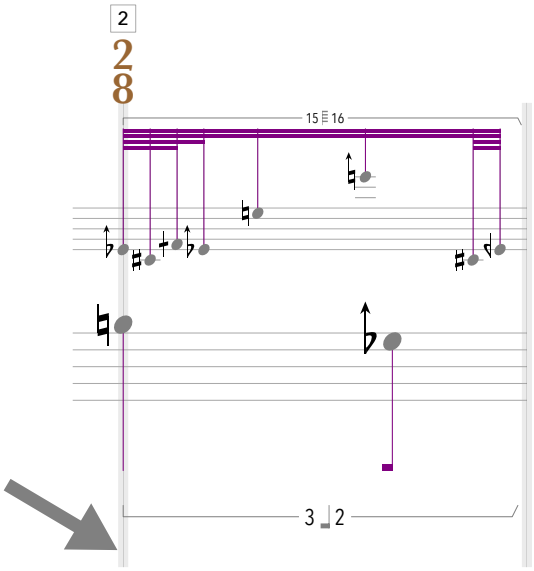

Figure 9.5: Barline.

MeasureNumber

The current implementation of measure numbers is simple, providing the one-based index of a given Measure. A MeasureNumber is a ViewNode that is organized in a MeasureNumberNode hierarchy, that is held ultimately within the System level of organization. Along with the TimeSignature and Barline, the MeasureNumber is centered exactly over the beginning of a Measure, in order to prevent the loss of the horizontal graphical representation of time.

9.2 Ensemble Organization

A single musician in an ensemble is modeled as a Performer in **denm**. This musician may be playing one or more instruments (e.g., double bass and voice). Each instrument may necessitate one or more strata of musical information (most commonly a Staff).

This model of ensemble organization is constructed to enable the dynamic creation of a View of the musical model of a piece (i.e., a Part) for a particular musician, where the musical material may warrant multiple representations. Coupled with the ViewNode model, these groupings of musical information may be dynamically shown or hidden as desired by the player. The level of notated detail of a given passage is then defined by the performer, optimized for their current stage of learning, rehearsing, or performing the piece.

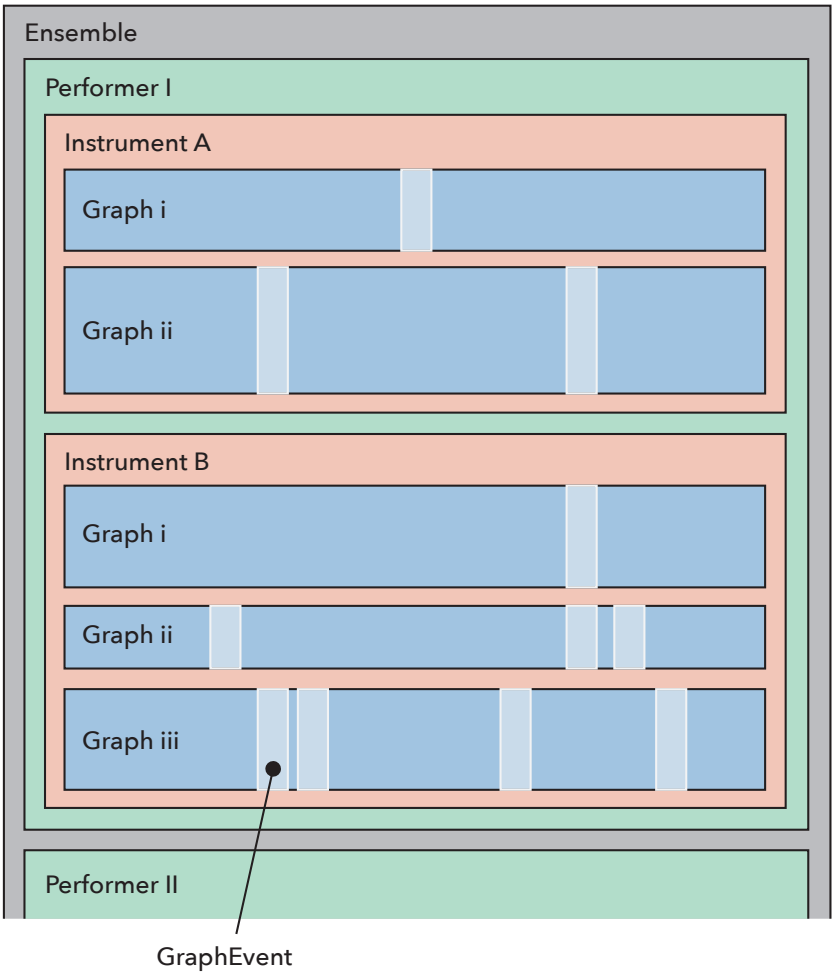

Figure 9.6: Ensemble Organization.

Fig. 9.6 shows an example of ensemble organization in **denm**. An ensemble may contain $1-n$ `Performers`. Each `Performer` may contain $1-n$ `Instruments`. Each `Instrument` may contain $1-n$ `Graph`. Each `Graph` may contain $0-n$ `GraphEvents`.

9.2.1 GraphEvent

A `GraphEvent` contains graphical information pertaining to a single representation of a musical event. For example, a triad of pitches arranged harmonically on a `Staff` would all be contained by the same `GraphEvent`.

9.2.2 Graph

The `Graph` is a subclass of `ViewNode`, that is contained by an Instrument. The `Graph` is the base class for all containers of placed musical information. The most common subclass of `Graph` is a `Staff`. In traditional music notation, noteheads and accidentals are placed vertically along five equidistantly-spaced horizontal lines. The placement of these objects specifies the pitch information stored in the musical model of a work. In other contexts, such as with string tablatures, finger or bow placement may be positioned along the y-axis of a more generalized `Graph` instance.

9.2.3 Instrument

The `Instrument` is a subclass of `ViewNode`, which is contained by a `Performer`. An `Instrument` is the graphical collection of `Graphs` pertaining to a single physical instrument (e.g., clarinet, contrabass, viola). Often, the notation for a single instrument will necessitate more than one `Staff` or other graphical representation of musical information.

9.2.4 Performer

The `Performer` is a subclass of `ViewNode`, that is contained by a `System`. In the graphical organization of the ensemble, a single musician is represented by a `Performer` class.

9.3 BeamGroup and its Organization

The `BeamGroup` is the graphical representation of a complete `DurationNode` tree structure (described in Sec. 7.4.)

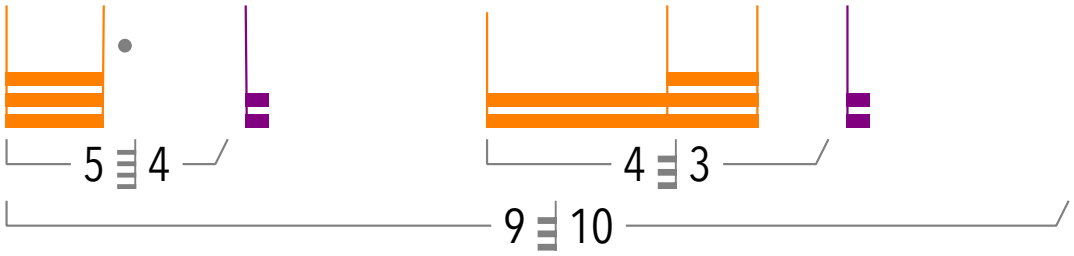

Figure 9.7: BeamGroup.

BeamGroup Event (BGEvent)

The graphical representation of a `leaf DurationNode`. Contains reference to `Stem`, and the `DurationNode leaf` with musical information in the form of `Components`.

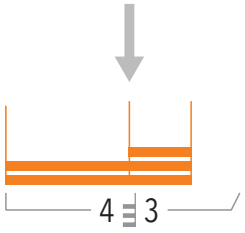

Figure 9.8: BeamGroupEvent.

BeamGroup Container (BGContainer)

The graphical representation of a `container DurationNode`.

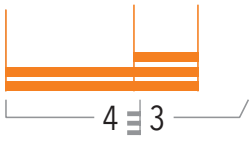

Figure 9.9: BeamGroupContainer.

BeamGroupStratum (BGStratum)

The graphical representation of multiple DurationNode tree structures comprising the musical information shown in a System.

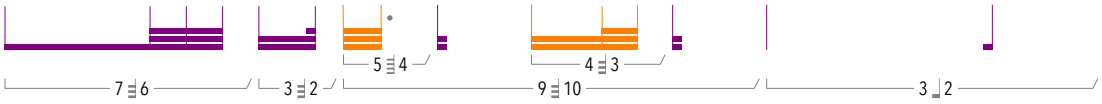

Figure 9.10: BeamGroupStratum.

9.4 Stems

A Stem is a subclass of Ligature that contains references to a BGEvent (discussed in Sec. 9.3) and a GraphEvent (discussed in Sec. 9.2.1). Stems are resized when the layout of their parent System is modified.

9.5 Music Notation Object Design

DynamicMarking

Figure 9.11: Dynamic Markings.

DynamicMarkings are collections of DynamicMarkingCharacters.

DynamicMarking Character (DMCharacter)

Figure 9.12: Currently implemented DynamicMarkingCharacters.

DynamicMarking Ligature (DMLigature)

Figure 9.13: DynamicMarkingLigature.

DynamicMarkingLigatures are automatically placed to connect a pair of DynamicMarkings.

Clefs

Figure 9.14: Clef Design.

The four clefs in Fig. 9.14 are treble, bass, alto, and tenor clefs. The minimalist design of these clefs minimize the horizontal space that they use. Clefs are colored red to maximize the visual contrast between the object and its environment.

Accidentals

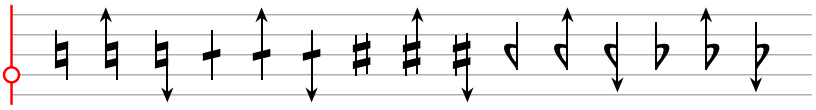

Figure 9.15: Accidental Design.

Accidentals are drawn programmatically, as opposed to being instances of glyphs from a font. The advantage to uniquely drawing each accidental is that small vertical adjustments can be made to individual components of the object (e.g.

body, column(s), arrow) in order to avoid collisions in a more dynamic fashion than is usually implemented in other music notation software (this accidental collision avoidance process is discussed in Sec. 11)

Chapter 10

Algorithms for Musical Analysis

10.1 Metrical Analysis

denm analyzes rhythms of any complexity. The result of this analysis provides a performer with an optimal manner in which to subdivide the rhythm. The graphical representation of this process is a `MetronomeGraphicNode`, which can be seen in Sec. 2.3. The metrical analysis process attempts to compare the actuation points of `DurationNode leaves` against a set of `MetricalAnalysisPrototypes`, which are sequences of two or three elements, each of which having a duple- or triple-beat `Duration`. The metrical analysis process is shown in Alg. 1. The output of the metrical analysis process is a `MetricalAnalysisNode`, which is a subclass of `DurationNode`, with the additional constraints that the sum of the relative durational values of the `children MetricalAnalysisNodes` is equivalent to the amount of `Beats` in the `Duration` of the given `MetricalAnalysisNode`.

Total durational values between and including 4 and 7 have a dedicated list of prototypes[1], each of which can be compared against any rhythm with a relative

[1]List of prototype sequences by relative durational sum:
$4 : (2, 2)$
$5 : (3, 2), (2, 3)$
$6 : (2, 2, 2), (3, 3)$
$7 : (2, 2, 3), (3, 2, 2), (2, 3, 2)$
$8 : (4, 4), (3, 3, 2), (2, 3, 3), (3, 2, 3) *$
$9 : (3, 3, 3), (4, 5), (5, 4) *$
$*$ Rhythms with relative durational sums of 8 and 9 are compared against all of these combinations shown here. If the combination with least syncopation contains only values of 2 or

durational sum of the same value, the process of which can be seen in Alg. 2.

Rhythms with relative durational sums of 8 and 9 are compared against all of these combinations shown here. If the combination with least syncopation contains only values of 2 or 3, a `MetricalAnalysisPrototype` is generated with `children` containing `Durations` of duple- and triple-values. In the case that the combination with least syncopation contains values > 3, internal `MetricalAnalysisNodes` are created with the `Durations` of these values. The original `DurationNode` array is partitioned at points determined by the combination. The `MetricalAnalysis` process is then applied for each partition, and `MetricalAnalysisNode` leaves with duple- and triple-value `Durations` are added to each of these internal `MetricalAnalysisNodes`.

Cuthbert and Ariza [23] apply their metrical analysis process to beaming in the score representation of rhythms. This strategy will be the model for continued development in **denm**, extending to cases of arbitrarily-deep nested-tuplet rhythms.

3, a `MetricalAnalysisPrototype` is generated with `children` containing `Durations` of duple- and triple-values. In the case that the combination with least syncopation contains values > 3, internal `MetricalAnalysisNodes` are created with the `Durations` of these values. The original `DurationNode` array is partitioned at points determined by the combination. The metrical analysis process is then applied for each partition, and `MetricalAnalysisNode leaves` with duple- and triple-value `Durations` are added to each of these internal `MetricalAnalysisNodes`.

Algorithm 1 Metrical Analysis

```
durNodes ←DurationNode.children
parent ←Root MetricalAnalysisNode
function ANALYZE(durNodes, parent)
    s ← durNodes.sum()
    if s = 1 then
        child ←MANode(beats: 2)
        ▷ subdivision level ∗ = 2
        ▷ add child to parent
    else if s <= 3 then
        child ←MANode(beats: s)
        ▷ add child to parent
    else if 4 <= s <= 7 then
        p ← prototypeWithLeastSyncopation
        for pp in p do
            child ←MANode(beats: pp)
            ▷ add child to parent
        end for
    else if 8 <= s <= 9 then
        p ← prototypeWithLeastSyncopation
        if p contains values > 3 then
            for pp in p do
                part ← durNodes partitioned at pp
                newParent ←MANode(beats: cc)
                analyze(part, newParent)
            end for
        end if
```

```
27:     else
28:         ▷ create array of all combinations
29:         ▷ of values 4 <= v <= 7 with sum of s
30:         c ← combinationWithLeastSyncopation
31:         for cc in c do
32:             part ← durNodes partitioned at cc
33:             newParent ←MANode(beats: cc)
34:             analyze(part, newParent)
35:         end for
36:     end if
37: end function
```

Algorithm 2 Syncopation

```
1: d ← durationNodes.cumulative()
2: ▷ e.g. [4, 5, 7] ← [4, 1, 2].cumulative()
3: p ← prototype.cumulative()
4: ▷ e.g. [2, 4, 7] ← [2, 2, 3].cumulative()
5: syncopation ← 0
6: function getSyncopation(d, p)
7:     if d[0] = p[0] then
8:         ▷ Rhythm beat falls on prototype beat
9:         ▷ No syncopation penalty added
10:        ▷ Adjust d and s accordingly
11:        getSyncopation(d, s)
12:    else if d[0] < p[0] then
13:        ▷ Rhythm beat falls before prototype beat
14:        ▷ Check if next duration falls on prototype beat
15:        if delayedMatch then
16:            ▷ No syncopation penalty added
17:        else
18:            syncopation ← syncopation + penalty
19:        end if
20:        ▷ Adjust d and s accordingly
21:        getSyncopation(d, s)
22:    else
23:        ▷ Rhythm beat falls after prototype beat
24:        ▷ Check if next prototype value falls on duration
25:        if delayedMatch then
26:            ▷ No syncopation penalty added
27:        else
28:            syncopation ← syncopation + penalty
29:        end if
30:        ▷ Adjust d and s accordingly
31:        getSyncopation(d, s)
32:    end if
33: end function
```

10.2 Automatic 1/8-tone Pitch Spelling

Effective pitch spelling is critical to the process of generating staff notation for music that is algorithmically composed or extracted from spectral analyses. Algorithms for pitch spelling within tonal musical contexts have been compared by Meredith [27] and Kilian [28]. The musical contexts that **denm** is most immediately supporting are rarely tonal. More often these musical contexts are microtonal. The preferences in the current tonally-based pitch spelling algorithms, however, are defined by establishing tonal center. The benefits of tonal-center-based pitch spelling are lost in atonal musical contexts. Primitive intervallic relationships are preserved objectively, rather than being subjected to the requirements of a tonal center. When spelling pitches in microtonal contexts, other pragmatics arise that influence the decision-making process.

The short-term goal of the pitch spelling process in **denm** is to spell microtonal polyphonic music up to the resolution of an 1/8-tone (48 equal divisions of the octave). In time, this process may be extended to accommodate other tuning systems. The development of this microtonal pitch spelling procedure is in process. Currently, verticalities consisting of any combination of resolutions (1/2-tone, 1/4-tone, 1/8-tone) are spelled correctly, though more rigorous testing is underway to verify this. Further development of this pitch spelling algorithm into polyphonic microtonal contexts will incorporate aspects of Cambouropoulos' shifting overlapping windowing technique [29].

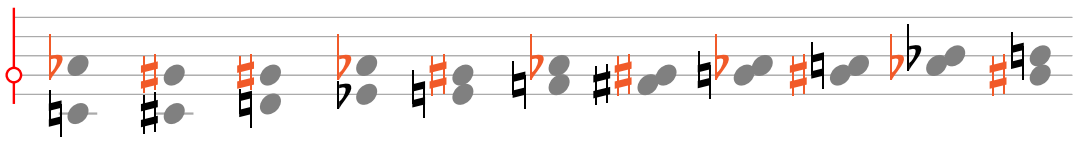

Figure 10.1: PitchDyadSpeller spelling 1/2-tone dyads contextually.

Fig. 10.1 shows the `MIDI Pitch` 68.0 (shown in orange) spelled contextually, depending on the other pitch present in a dyad. This demonstrates the underlying priority in the pitch spelling process of **denm** called step-preservation, where

augmented and diminished are discouraged in favor for perfect, major, and minor intervals.

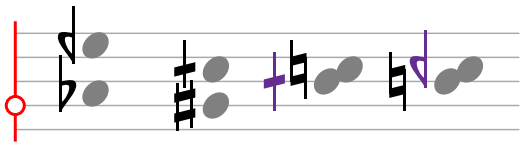

Figure 10.2: PitchDyadSpeller spelling 1/4-tone dyads contextually.

Fig. 10.2 shows dyads consisting of a combination of 1/2-tone and 1/4-tone pitches spelled. Identical direction of the `Coarse` values of the spellings is prioritized: if one `Pitch` is spelled as a 1/4-tone sharp, the other `Pitch` should be spelled as a sharp, not a flat.

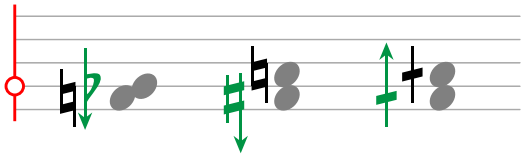

Figure 10.3: PitchDyadSpeller spelling 1/8-tone dyads contextually.

Fig. 10.3 shows the `MIDI Pitch` 65.75 spelled in each of three different possible ways, depending on the other `Pitch` in a dyad.

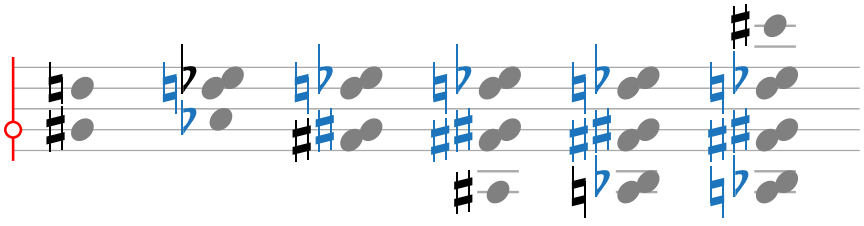

Figure 10.4: PitchVerticalitySpeller spelling 1/2-tone verticalities contextually.

Fig. 10.4 shows an accumulation of 1/2-tone `Pitches` into more complex harmonies. As a new `Pitch` is added to the harmony, each `Pitch` in the harmony is reconsidered for spelling.

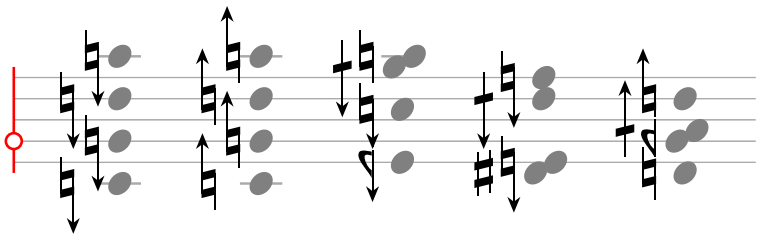

Figure 10.5: PitchVerticalitySpeller spelling 1/8-tone verticalities contextually.

Fig. 10.5 shows a set of verticialities of `Pitches` that consist of combinations of resolution (1/2-tone, 1/4-tone, 1/8-tone). When there are multiple `Pitches` in a harmony that have an 1/8-tone resolution, and therefore have an arrow in their graphical representation, a consistency of arrow direction is enforced. By enforcing equivalence of arrow direction, the resolutions of intervals of each dyad are made explicit.

Chapter 11

Accidental Collision Avoidance

The stacking of accidentals has been compared extensively by Spreadbury [30]. Accidental stacking in traditional typesetting software uses quite conservative rules to dictate the shifting of accidentals into new columns. In contexts where music spacing is quite loose, this technique works acceptably well. More complex pitch contexts, however, often require a considerable amount of manual re-positioning of accidentals.

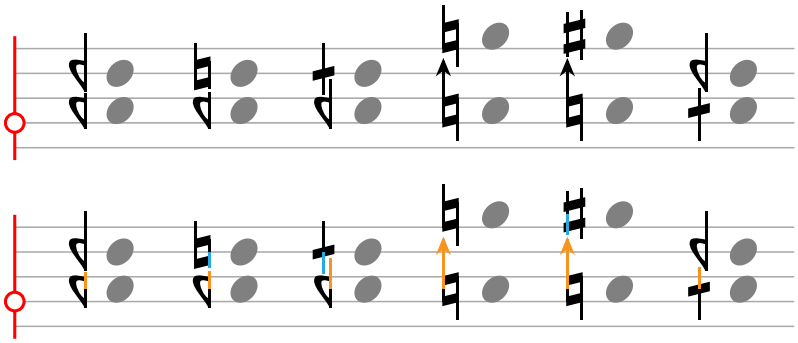

Figure 11.1: Accidental Collision.

Fig. 11.1 shows dyads with accidentals that traditionally collide. For **denm**, individual components of the colliding accidentals have been modified to avoid a lower-level collision. The bottom staff shows the components adjusted with cyan and orange colors. Stems are shortened and arrows are moved as necessary. In the worst case scenario, where a collision cannot be avoided via movement of these

components, the accidentals are stacked in the traditional fashion. By extending the sense of accidental stacking to a more local concern, a significant amount of horizontal space can be saved.

Bibliography

[1] S. Schick, "Developing an interpretive context: Learning brian ferneyhough's bone alphabet," *Perspectives of new music*, pp. 132–153, 1994.

[2] B. Ferneyhough, "Sisyphus redux." C.F. Peters, 2012.

[3] T. Papatrechas, "Thoracic asphyxiating dystrophy." Self Published, 2015.

[4] "forscore: the music reader for ipad," 2015.

[5] F. Donatoni, "Nidi." Universal Music Publishing, 1992.

[6] B. Ferneyhough, "Mnemosyne." C.F. Peters, 1986.

[7] P. Archbold, "Performing complexity," 2011.

[8] B. Oztan, G. Sharma, and R. P. Loce, "Quantitative evaluation of misregistration-induced color shifts in color halftones," 2005.

[9] B. Pimentel, "Fingering diagram builder."

[10] D. Crockford, "The application/json media type for javascript object notation (json)," 2006.

[11] Hexler, "Touchosc."

[12] H. Wulfson, G. D. Barrett, and M. Winter, "Automatic notation generators," in *Proceedings of the 7th International Conference on New Interfaces for Musical Expression*, NIME '07, pp. 346–351, 2007.

[13] H.-W. Nienhuys and N. Jan, "Lilypond, a system for automated music engraving," *Colloquium on Musical Informatics (XIV CIM 2003)*, May 2003.

[14] K. Renz, *Algorithms and Data Structures for a Music Notation System based on GUIDO Music Notation*. PhD thesis.

[15] T. Baca, J. Oberholtzer, and V. Adan, "In conversation."

[16] D. Psenicka, "Fomus, a music notation software package for computer music composers," in *International Computer Music Conference*, (San Francisco), pp. 75–78. San Francisco, International Computer Music Association., 2007.

[17] J. Bresson, C. Agon, and G. Assayag, "Openmusic: Visual programming environment for music composition, analysis and research," in *Proceedings of the 19th ACM international conference on Multimedia*, pp. 743–746, ACM, 2011.

[18] A. Agostini and D. Ghisi, "What is bach?."

[19] M. Laurson, M. Kuuskankare, and V. Norilo, "An overview of pwgl, a visual programming environment for music," *Computer Music Journal*, vol. 33, pp. 19–31, March 2009.

[20] M. Kuuskankare and M. Laurson, "Expressive notation package-an overview.," in *ISMIR*, 2004.

[21] N. Didkovsky and P. Burk, "Java music specification language, an introduction and overview," in *Proceedings of the International Computer Music Conference*, pp. 123–126, 2001.

[22] W. Burnson, *Introducing Belle, Bonne, Sage*. Ann Arbor, MI: MPublishing, University of Michigan Library, 2010.

[23] M. S. Cuthbert and C. Ariza, "music21: A toolkit for computer-aided musicology and symbolic music data," vol. 11, (Utrecht, Netherlands), pp. 637–642, International Society for Music Information Retrieval, August 2010.

[24] D. Fober, Y. Orlarey, and S. Letz, "Inscore: An environment for the design of live music scores," *Proceedings of the Linux Audio Conference - LAC 2012*, 2014.

[25] G. D. Barrett and M. Winter, "Livescore: Real-time notation in the music of harris wulfson," *Contemporary Music Review*, vol. 29, no. 1, pp. 55–62, 2010.

[26] C. Hope and L. Vickery, "Visualising the score: Screening scores in realtime performance," 2011.

[27] D. Meredith, "Comparing pitch spelling algorithms on a large corpus of tonal music," in *Computer Music Modeling and Retrieval*, pp. 173–192, Springer, 2005.

[28] J. Kilian, *Inferring score level musical information from low-level musical data*. PhD thesis, TU Darmstadt, 2005.

[29] E. Cambouropoulos, "Pitch spelling: A computational model," *Music Perception*, vol. 20, no. 4, pp. 411–429, 2003.

[30] D. Spreadbury, “Accidental stacking.”