MOCKINGBIRD: AN INTERACTIVE COMPOSER'S AID
by
JOHN TURNER MAXWELL III

SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREES OF

BACHELOR OF SCIENCE
and
MASTER OF SCIENCE

at the
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
January 1981

© Copyright John Turner Maxwell III, 1981

The author hereby grants to M.I.T. permission to reproduce and to distribute copies of this thesis document in whole or in part.

Signature of Author ____________________________

Department of Electrical Engineering and Computer Science, January 15, 1981

Certified by __________________________________
Edward Fredkin
Thesis Supervisor (Academic)

Certified by __________________________________
Severo Ornstein
Thesis Supervisor (VI-A Cooperative Company)

Accepted by __________________________________
Arthur C. Smith
Chairman, Department Graduate Committee

ARCHIVES
MASSACHUSETTS INSTITUTE
OF TECHNOLOGY

APR 30 1981

LIBRARIES
Mockingbird:
An Interactive Composer's Aid

by

John Turner Maxwell III

Submitted to the Department of Electrical Engineering and
Computer Science on January 15, 1981 in
partial fulfillment of the requirements for the Degrees of
Bachelors of Science and Masters of Science in
Computer Science

ABSTRACT

The subject of this thesis is Mockingbird, a computer-based music editor designed to aid musicians in composing music. The goal of the project was to produce a music editor that would permit a composer to capture musical ideas, edit them, and turn them into a score with less effort than is required by present means. We are focused on the composer's needs, not the publisher's. We especially want to be able to handle reasonably complex modern music, since this more involved style of composition drives us to a deeper understanding of the needs of all composers.

Mockingbird runs on a high-speed personal computer called a Dorado which has an eight and a half by eleven inch raster-scan display and a pointing device called a mouse. Music is edited by manipulating a representation of a score that appears on the display. It can be recorded and played back through an attached Yamaha CP-30 electronic synthesizer.

In this thesis we discuss some of the problems that composers face, a novel model for music, how Mockingbird works, and our conclusions about the usefulness of this type of composer's aid. We also step through an example editing session using Mockingbird.

Academic Thesis Supervisor:    Edward Fredkin
Title:    Professor of Computer Science and Electrical Engineering

Company Thesis Supervisor:    Severo Ornstein
Title:    Manager of Computer Organization, Xerox Palo Alto Research Center
ACKNOWLEDGEMENTS

Mockingbird would not have been possible without the help of a great number of people at PARC and at MIT.

I would first like to thank MIT and PARC for allowing me to be part of the VI-A program, which enabled me to attempt this work.

I owe the most thanks to Severo Ornstein and Will Crowther for their moral and intellectual help. Most of the ideas expressed in Mockingbird were influenced by long discussions with them. They were also very enthusiastic, and kept me going when things became difficult.

I also owe thanks to Gene McDaniel, Doug Wyatt, and Mike Overton for contributing their expertise. Gene McDaniel produced the microcode that allows Mockingbird to record and play music on the synthesizer. Doug Wyatt designed the music fonts used by Mockingbird for displaying music on the screen. Mike Overton installed the hardware that Severo Ornstein had designed to interface the synthesizer with the Dorado computer.

I would especially like to thank Austin Henderson for his help as the VI-A coordinator and for proofreading this thesis. His commentaries were always useful and thorough. Because of his discussions, several things that were not very clear have become more so.

Finally, I would like to thank all of the people at PARC who encouraged me by showing enthusiasm for my project. They made me feel that the work I was doing was worthwhile.
# Table of Contents

## 1.0 Introduction
1.1 The Composer's Problems .......................... 9
1.2 General Approach .................................. 10
   1.2.1 biased towards piano music .................. 10
   1.2.2 an interactive editor, not an automatic recognizer 10
1.3 What Mockingbird Does .......................... 11
   1.3.1 capturing music .............................. 11
   1.3.2 displaying music ............................. 11
   1.3.3 manipulating music ......................... 11
   1.3.4 performing music ............................ 11
1.4 What Mockingbird Does Not Do .................. 12
   1.4.1 computer generated composition ............ 12
   1.4.2 high quality performances .................. 12
   1.4.3 sound synthesis .................................. 12
   1.4.4 publishable scores ........................... 12
   1.4.5 optical character recognition ............... 13
   1.4.6 sound recognition ............................ 13

## 2.0 Background
2.1 Three Representations for Music ............... 14
   2.1.1 Physical Representation .................... 14
   2.1.2 Logical Representation ...................... 16
      2.1.2.1 notes .................................. 17
      2.1.2.2 rests .................................. 17
      2.1.2.3 embellishments ......................... 17
      2.1.2.4 voices ................................ 18
      2.1.2.5 beams, phrases, measure lines, and chords 18
      2.1.2.6 sectional attributes .................... 19
   2.1.3 Graphical Representation .................. 19
      2.1.3.1 logical objects ......................... 19
      2.1.3.2 staffs and clefs ....................... 20
      2.1.3.3 spacing and placement .................. 20
      2.1.3.4 stem direction and beam tilt .......... 20
2.2 Mockingbird's Model of Music ................ 21
   2.2.1 A Sequential Model of Music ............... 21
   2.2.2 The Advantages of a Sequential Data Structure 22
      2.2.2.1 one data structure for all three domains 22
      2.2.2.2 behaves well under inconsistencies .... 22
      2.2.2.3 procedures are easy to code .......... 23
      2.2.2.4 time critical actions are sequential 24
2.3 why not an automatic recognizer? .............. 24
   2.3.1 Determining Note Durations ................. 24
   2.3.2 Separating a Piece into its Voices .......... 26
   2.3.3 Grouping Notes with Beams and Phrase Marks 27
3.0 EXAMPLE
4.0 THE USER’S VIEW
4.0.1 Goals
4.0.2 Metaphors
  4.0.2.1 functionality
  4.0.2.2 form
4.1 USER ACTIONS
  4.1.1 Utilities
    4.1.1.1 recording
    4.1.1.2 playing
    4.1.1.3 filing in and out
    4.1.1.4 quitting
  4.1.2 View Manipulations
    4.1.2.1 scrolling and scaling the window
    4.1.2.2 choosing the sheet
    4.1.2.3 choosing the key
    4.1.2.4 showing the voices
    4.1.2.5 showing the event lines
    4.1.2.6 showing the durations
    4.1.2.7 playing the music
    4.1.2.8 choosing the domain
  4.1.3 Data Manipulations
    4.1.3.1 manipulating the sequence of events
    4.1.3.2 adding or subtracting notes
    4.1.3.3 grouping notes into events, chords, beams, and tuplets
    4.1.3.4 changing note attributes
    4.1.3.5 changing event attributes
  4.1.4 Miscellaneous
    4.1.4.1 heuristic actions
    4.1.4.2 constraint checking
    4.1.4.3 layout
4.2 THE USER INTERFACE
  4.2.1 Basic Tools
    4.2.1.1 screen
    4.2.1.2 keyboard
    4.2.1.3 mouse
    4.2.1.4 cursor
  4.2.2 Selection
    4.2.2.1 note selection
    4.2.2.2 section selection
  4.2.3 Commands
    4.2.3.1 keyboard commands
    4.2.3.2 mouse commands
  4.2.4 Graphical Commands
    4.2.4.1 moving notes
    4.2.4.2 inserting notes
    4.2.4.3 the special menu
5.0 THE PROGRAMMER’S VIEW

5.1 DESIGN PHILOSOPHY
5.1.1 Separating Functionality and Efficiency
5.1.2 Separating Orthogonal Issues By Means Of Interfaces
5.1.3 Tools First, Then Heuristics

5.2 DATA STRUCTURE
5.2.1 Storage
5.2.2 The Piece
5.2.2.1 notes
5.2.2.2 chords
5.2.2.3 beamed sets
5.2.2.4 tuplets
5.2.2.5 events
5.2.2.6 voices
5.2.2.7 the sequence of events

5.2.3 The Sheet
5.2.3.1 translating between coordinate systems
5.2.3.2 determining staffing
5.2.3.3 determining accidentals

5.3 THE GRAPHICAL VIEW
5.3.1 Graphics Package
5.3.1.1 basic commands
5.3.1.2 concept of context
5.3.1.3 the music font
5.3.2 Displaying Music Objects
5.3.2.1 sheet
5.3.2.2 notes
5.3.2.3 chords
5.3.2.4 beams
5.3.2.5 key signatures
5.3.2.6 time signatures

5.3.3 Layout
5.3.3.1 physical domain
5.3.3.2 logical domain

5.4 THE AUDIBLE VIEW
5.4.1 The Synthesizer and Microcode
5.4.1.1 the synthesizer
5.4.1.2 the microcode
5.4.2 The Player
5.4.2.1 physical music in the physical domain
5.4.2.2 logical music in the logical domain
5.4.2.3 problems caused by switching domains

5.4.3 The Recorder

6.0 CONCLUSIONS

6.1 LEFT TO DO
6.1.1 Musical Items
6.1.1.1 embellishments
6.1.1.2 performance cues
6.1.1.3 ties and phrase marks
6.1.1.4 spelling
6.1.1.5 sectional staffing
6.1.1.6 graphical adjustments
6.1.2 Program Items
   6.1.2.1 multiple windows
   6.1.2.2 copying structures
   6.1.2.3 undo

6.2 ADVANTAGES OF MOCKINGBIRD
   6.2.1 Faster Capture
   6.2.2 Easy Editing
   6.2.3 Better Error Detection

6.3 A TEACHER'S AID - ANOTHER APPLICATION
   6.3.1 Reading
   6.3.2 Understanding Music and Music Notation
   6.3.3 Performing

6.4 IMPLICATIONS
LIST OF FIGURES

2.1. Leo Ornstein, Piano Sonata #4, page 1 28
2.2. Leo Ornstein, Piano Sonata #4, page 31 29
2.3. Bach-Busoni, Chaconne in D minor, page 17 30
2.4. Resolving Inconsistencies in the Piece 23
2.5. Example Events 31
2.6. Problems With Automatic Recognition 32
3.1-3.40. Snapshots of an Editing Session 34-73
3.41. Chopin, Third Ballade, first nine measures 74
4.1. Mockingbird as it appears on the screen 91
4.2. List of Commands 92
5.1. Model of the Data Structure 115
1.0 INTRODUCTION

The subject of this thesis is Mockingbird, a computer-based music editor designed to aid musicians in composing music. The goal of the project was to produce a music editor that would permit a composer to capture musical ideas, edit them, and turn them into a score with less effort than is required by present means. We are focused on the composer's needs, not the publisher's. We especially want to be able to handle reasonably complex modern music, since this more involved style of composition drives us to a deeper understanding of the needs of all composers.

Mockingbird was developed over a period of five months in 1980 at Xerox's Palo Alto Research Center. It is written in Mesa, a strongly typed language developed by the Computer Science Lab at PARC, and runs on a Dorado with an attached Yamaha CP-30 76 note synthesizer. The Dorado is a high performance personal computer with a eight and a half by eleven inch bit map display and a pointing device known as a "mouse".

The thesis discusses our representation for music, an example use of Mockingbird, how it appears to the user, and how it appears to the programmer. At the end we evaluate our work and discuss where we might go from here. The remainder of this introduction is concerned with defining the problem more carefully and showing where our work fits in the realm of music.

1.1 The Composer's Problems

One of the most serious problems a composer faces is that of capturing musical ideas quickly before they fade and are lost. Many composers do not build their compositions a brick at a time, but instead get an inspiration for a chunk of theme or melodic line which comes into their head all at once. Ideas come rapidly, and must be scribbled down before they are pushed out of the way by new ideas.

Some composers try to get around these problems by using a tape recorder as they compose on an instrument. After they have generated a number of ideas, they edit the tape to produce a completed piece. This has the difficulties that finding a particular idea on a tape is time consuming since tapes can only be accessed sequentially, and that editing a tape is a slow and painful process requiring the composer to physically cut and paste pieces of tape. We tried to avoid both of these problems as we designed Mockingbird.
As critical as it is, the process of capturing ideas is only the first step for the composer. An important and time consuming part of composition involves editing music: cutting and pasting, building transitions between ideas, radically reworking sections to improve them, and laying out a reasonable approximation to a score. We would like to provide the composer with a tool that aids him in all of these areas.

1.2 General Approach

From the outset we have made a number of assumptions concerning the form of the editor and the needs of a composer. Furthermore we have adopted some fairly firm guidelines in our general approach to the problem. We discuss each of these here.

1.2.1 biased towards piano music

Because the piano is the most general instrument for playing multiple part music, many musicians use the piano while composing even though other instruments may eventually play the music. Thus composers who write orchestral or chamber music often develop the basic themes on a piano. For this reason, and because keyboard instruments are easy to connect to a computer, we use an electronic synthesizer for recording and playing music. Mockingbird is thus heavily biased towards keyboard music.

1.2.2 an interactive editor, not an automatic recognizer

An automatic recognizer would transform music as played on the synthesizer directly into a complete score. We believed that such a project would be extremely difficult: except for trivial cases with almost no chance of success in general. (See section 2.3.) Therefore, instead of going down this path, we decided to provide the tools that would allow the composer to perform the transformation himself. This meant building an interactive music editor in which musicians could work with the music in forms with which they were familiar. For this reason, our editor can 1) display music in standard music notation, 2) play music upon command, and 3) accept music from a (synthesizer) keyboard.

1.3 What Mockingbird Does

To be helpful to the musician, the editor should be able to capture, display, manipulate, and perform musical ideas in a form he understands. In the following sections, we discuss each of these activities and how Mockingbird accomplishes them.
1.3.1 capturing music

The musician must have some means of transmitting ideas to the editor quickly, naturally, and faithfully. A very quick way that a musician can express an idea is by performing it. There are two ways to enter music in Mockingbird: by playing it on the synthesizer keyboard directly or by adding notes to the score manually. In the first, Mockingbird observes the synthesizer keystrokes and records their occurrence and duration. This allows the composer to enter music quickly and naturally in a form much like an old fashioned pianoroll. In the second, the composer adds notes by pointing at a position in the score with the mouse and commanding the editor to deposit a note of a specified value. This second method is less facile than the first, but is useful for making small corrections to the score.

1.3.2 displaying music

Mockingbird can display music in a number of different ways. In the most primitive form, the music entered from the synthesizer keyboard looks much like a pianoroll. As the user edits this pianoroll, it begins to appear in a form that the user recognizes. In the final stages, the music looks like a standard score. We discuss how Mockingbird produces these displays in section 5.3.

1.3.3 manipulating music

Before the editor can be used to manipulate musical ideas, an internal representation must be established. We decided that the editor would use a representation that matched our idea of the musician’s internal representation. The musician manipulates the piece of music through the display with a number of commands whose functions match our intuitions about musical operations. These commands are discussed in section 4.1.

1.3.4 performing music

The editor performs a piece for the musician by interpreting and playing it. The first part involves translating the editor’s representation for music into manipulations of the synthesizer, and the second involves causing the synthesizer to produce sound. Mockingbird produces a reasonable interpretation of the piece, ignoring the nuances necessary for a "great" performance. This interpretation is then fed to the synthesizer, which plays it.

1.4 What Mockingbird Does Not Do

A number of things have been suggested as improvements to our editor. Although
interesting, most of these are not pertinent to our primary goal of aiding the composer. We list them here for two reasons: first, to provide contrast with, and more clearly define, what we are attempting to do, and second, to explain why we have chosen to ignore these areas.

1.4.1 computer generated composition

Computer generated composition uses the computer directly as a composer, generating new and interesting pieces algorithmically. We are not interested in this area since our goal is to aid the composer, not replace him.

1.4.2 high quality performances

A high quality performance is one that handles all of the subtle nuances of music, dealing with such things as timing, tempo, and loudness. The goal of such work would be to produce a performance that people would enjoy listening to qua performance. This is closer to our goal, since the editor does perform music for the musician. However, the performance needs only to be good enough for the musician to understand: we are not committed to producing performances that the musician or an audience will like.

1.4.3 sound synthesis

Another aspect of a high quality performance would be generating instrumental sounds that people would enjoy listening to. The computer is a versatile tool that can be used to generate altogether new sounds or to mimic those of the classical instruments. Again, we are concerned with the musician's comprehension, not his enjoyment of the performance. The only thing we ask is that the instrument's sounds not irritate and thus distract him.

1.4.4 publishable scores

Mockingbird has to be able to produce a score of some sort for the composer, but that score does not have to be of publishable quality. Mockingbird's score is generally a reasonable representation of the piece which any musician could read. However, it usually falls short of the high standards required for publication. We are not too concerned with this failing, since publishing is not part of our main goal. Publishers expect to receive scores that have to be worked on some before they can be published.

1.4.5 optical character recognition

Currently in Mockingbird, music can only be entered either by playing it on a piano-style keyboard or by using a graphical device called a mouse. To enter a piece that had already been
published, one could play it in, but much of the information contained in the score would be lost in the process. (See section 2.3 for an explanation.) It would be nice if the editor could read scores directly via some kind of optical scanner so that finished scores could be modified. However our main concern is with aiding the composer to capture new, unpublished pieces.

1.4.6 sound recognition

Finally, one might consider trying to capture scores by listening to the sounds while they are played. This would involve processing sound waves to extract pitch and timing information, which in turn would involve isolating the sounds of different instruments. Although this would fit in well with our goals, we believe this to be an extremely difficult task which would consume all of our attention and thus deflect us from our main goal. We therefore chose to limit ourselves to a keyboard instrument where the keys struck can be observed directly by the computer. We believe that most composers will find a keyboard instrument to be adequate.
2.0 BACKGROUND

In this chapter we discuss our models for music. The first section deals with three representations that we feel are important to understanding music. This classification is not the only way that music could be broken up, but we have found it useful. The second section talks about a model for music that to some extent incorporates all the representations discussed in the first. This model is fundamental to Mockingbird's understanding of music. The last section talks a little bit about why we chose to produce an amanuensis or aid rather than a program that attempts to translate directly from pianoroll to score.

2.1 THREE REPRESENTATIONS FOR MUSIC

We will consider three representations for music corresponding to the three domains which we call physical, logical, and graphical. When we speak of a domain we are talking about how something is viewed. When we speak of a representation we are talking about how something is modeled. This distinction will be important because we may want to discuss how a logical representation is viewed in the physical domain, for example.

The physical domain views music as sound waves or manipulations of a musical instrument. An example of a representation for the physical domain is a pianoroll which encodes such manipulations. The pianoroll captures the exact rhythm and phrasing that the performer used, but it contains no information about the relationships and structural elements of the music played.

The logical domain views music more as we believe the composer sees it, i.e. represented "logically" as a collection of notes with specially defined values (wholes, halves, quarters, etc). There is also information about the relationships between notes: how the notes are grouped into chords, beams, and phrases. Furthermore, the piece is divided into voices. So the logical domain is heavily concerned with structure and relationships.

The graphical domain views music as a score to be presented visually. Music may be represented graphically as a collection of marks on a sheet of paper. Most of these marks have a corresponding meaning in the logical domain. For example, a mark made up of a solid oval with a vertical line and a single attached flag corresponds to the logical domain's notion of an "eighth note". In addition there are aesthetic relations that are quite independent of the logical
domain since one of the goals of a good graphical representation is to present the material in an uncluttered view (e.g. to employ suitable spacing and placement).

An analogous division occurs in speech processing. Words are spoken (physical), understood (logical), and printed (graphical). A tape recorder can capture and reproduce speech with remarkable fidelity by representing it as a continuous (physical) waveform. It would take a much more sophisticated system than a tape recorder to parse or understand the words. A knowledge based system that attempted to understand speech would have to employ a different (logical) representation for it which incorporated the semantics. Finally, a document preparation system typically manipulates the (graphical) marks that we use to represent words without ever understanding them. Many of the elements in the graphical domain (e.g. justification, line breaks, layout) have no correspondence to elements in the other domains, and so a third representation is called for.

Most word processors attempt to work only in the graphical domain; our music editor works in all three. It must be able to capture music from a keyboard and play it back (physical domain), manipulate music by assigning and modifying structures (logical domain), and generate scores (graphical domain).

2.1.1 Physical Representation

An example of a physical representation in the real world is a pianoroll. Pianorolls are long pieces of paper with holes. Each string of holes indicates a pitch and a duration. The starting position of the string of holes determines when the note will be played. For convenience, we often refer to a particular physical representation as a pianoroll even though the information will be encoded as bits in the computer's memory rather than as holes in a roll of paper.

Mockingbird's physical model of music is a collection of pianoroll notes. Each note has four attributes: time, pitch, duration and loudness. These completely specify how a note is played. The time of each note is measured in some real-time units (seconds, milliseconds, or whatever) and specifies the instant that the note is to commence. The pitch gives the particular key that is to be struck. There are 88 pitches; one for each of the keys on a standard piano. The duration tells how long the key is to be held before it is released. Finally, the loudness indicates the volume at which the note is to be played. (We chose to ignore loudness information in Mockingbird since we felt that it did not add much to the musician's
understanding of the piece and was difficult to infer for music which had not come fairly directly from the synthesizer keyboard. All of Mockingbird's music is therefore played at uniform loudness.)

2.1.2 Logical Representation

The logical representation is based on our model of how a composer understands music. We have found that it is easier for the user to manipulate an abstract logical representation such as this than to manipulate marks on the display. This is because the logical representation "understands" the basic elements of music in much the same way that a musician might.

We call an instance of a logical representation a piece. A piece is a collection of interdependent voices broken up into measures. Music may be made up of more than a single melody; it generally has a number of interdependent ideas that make up a piece's melody, harmony and counterpoint. We call these ideas "voices". Our concept of a voice is not restricted to a melodic line: a voice may have chords as well as single notes (or rests) sometimes grouped by beams and phrases. A voice is a complete musical idea somewhat akin to a part.

There are also a number of attributes (such as time and key signatures) that apply to sections of the piece. The various objects that make up a piece are discussed below.

2.1.2.1 Notes

A logical note has the same attributes as a physical note: time, pitch, duration, and loudness. However, in the logical domain these attributes are discrete instead of continuous. Both time and duration are measured in beats. As normally used in music, the term "beat" defines the time unit of the rhythmic pulse of a piece and appears as the denominator in the time signature. Thus in 3/4 time, each beat is a quarter note (and three such beats form a measure). Our use of the term "beat" will be somewhat more general and will stand for a logical time unit whose purpose is to provide an abstract time grid. Specifically, the term "beat" will mean one whole note or whole rest equivalent. Thus two measures of 3/4 time will contain, in our terms, one and a half "beats" (2 * 3/4) as opposed to the traditional definition which would assign it six beats. The reader should keep this special use of the term in mind throughout the remainder of this document.

The time of a logical note is measured in beats from the beginning of the piece. The time is not kept explicitly however: it is implied by the order of the notes. The time of a note is the
sum of the previous note's time and its duration. Determining the time of a note is complex when there are several voices since "previous" becomes ambiguous. We discuss this complication in section 5.4.2.2.

The duration of a note is a fraction of the basic logical time unit (i.e. of the beat). The standard durations are binary fractions: whole, half, quarter, eighth, sixteenth, thirty-second, and sixty-fourth. Non-standard durations can be obtained with dots, tied notes, or tuplets. A dot extends the duration of a note by a half. Thus a dotted quarter note is equal to 3/8 (1/4 x 3/2). Ties allow durations to be expressed as sums of other durations. This means that a 5/16 note can be obtained with a quarter note tied to a sixteenth note. Finally, non-binary fractions can be obtained with tuplets. A tuple is a collection of some notes with a nominal duration sum of n beats which are to fit in a slot for m beats (n≠m). For example, a triplet is made up of three notes that fit in the space that would normally be occupied by two notes. Each note could be thought of as having a duration of two-thirds of its normal value. The notes of a tuplet do not have to have the same duration. For example, a half note coupled with a quarter note makes a perfectly legal triplet. The important thing is the sum of the durations.

2.1.2.2 rests

A rest is place holder. It is like a note without pitch or loudness and indicates a place in the score where nothing is to be played for a specified number of beats. Rests have the same durations as notes (whole, half, etc. (optionally dotted)), but they are displayed differently.

2.1.2.3 embellishments

Embellishments are special notes or attributes that are used as a shorthand for fancy styles or techniques of playing. Grace notes, trills, mordents, and rolled chords are all embellishments. Conceptually, grace notes do not occupy any logical time although they do have a logical duration. They are played by stealing time from their neighbors. Trills and mordents are a shorthand for particular types of sequences of notes. The performer can execute the sequence if he knows the note it begins on. Rolled chords are a particular technique for playing a chord where the notes are played rapidly one after another rather than simultaneously.

2.1.2.4 voices

In this thesis we use the term voice to mean something slightly different than it traditionally means. A voice, for us, is a succession of notes and/or chords which fit together within a piece's
rhythmic structure in such a way as to form a complete and consistent set over some part of the piece. This meaning is best understood by looking at an example.

Let us examine figure 2.1. In the first measure there are two voices; the grace notes, chords, and rest in the right hand form one voice and the arpeggio in the left hand forms the other. Note that the first note in the second septuplet forms a part of both voices. Such sharing is not uncommon.

Now look at the fifth measure. Here there are three voices; the arpeggio in the left hand, the two quarter note elements, and the descending eighth notes. This same voicing extends into the following measure but in the middle of that measure the right hand breaks up briefly into three separate voices (making a total of four voices). In the seventh and eighth measures there are again three voices, but in the ninth there are only two: the arpeggio and the melodic line's chords. The "inner voice" reappears in the middle of the following (tenth) measure.

Thus, a voice does not necessarily run through an entire piece of music; it may start or stop abruptly or have long gaps. Some voices have very short sections that appear intermittently. Where a voice is present it precisely fills the rhythmic structure. Small gaps are filled with rests; during large gaps, the voice simply disappears. The progression of a voice is indicated in a score by a number of clues such as stem direction, chording, staffing, and note durations. When these implicit indications are insufficient to define a voice, the composer will use explicit indications such as drawing lines between notes in the same voice.

2.1.2.5 beams, phrases, measure lines, and chords

Each of the above items group notes into conceptual units. The groupings serve two purposes: indicating important semantic units in the piece, and (thereby) making it easier for the reader to keep track of where he is. We believe that the structure given by these items reflects structure in the composer's mind. These groupings affect how a musician understands and plays the piece.

Chords group notes "vertically" into harmonic structures. Sometimes chords will correspond to the hand divisions used on a keyboard instrument. Other times they will indicate a group of notes all in the same voice.

Beams and phrases group notes (and/or chords) "horizontally". Beams are used for smaller structures, phrases for larger. They both indicate groups of items that should be considered
units in the larger structure of the piece.

Measure lines are used as markers in music; they help the musician find his way around in a piece. Measure lines group notes horizontally into units with a regular number of beats. (This number may change from measure to measure, but the musician is always aware of the current number.) Measures are not inviolable; ties, beams, and sometimes even note durations traverse the boundary between two measures (See figure 2.2.5).

2.1.2.6 sectional attributes

There are a number of attributes that apply to sections of a piece rather than to individual notes. These attributes are: key signatures, time signatures, dynamics, and tempo markings. Keys signatures indicate changes in the key that are to apply over the following section. Time signatures tell the number and type of beats that are found in the following measures. Dynamics markings (e.g. forte or pianissimo) specify changes in the loudness that a section is played. Tempo markings (e.g. andante or accelerando) change the tempo at which a section is to be played.

2.1.3 Graphical Representation

The graphical representation for music, called "standard music notation", is that which appears on a sheet of music. It has a representation for all of the logical information mentioned above, plus information peculiar to the score's visual requirements: staffs, clefs, stem direction, beam tilts, spacing, and placement. This section will concentrate on the additional aspects of the graphical representation.

In principle, the graphical representation can be derived from the logical using a set of conventions that embody the goal of producing a dense but uncluttered picture which clearly depicts the logical structures of a piece. In practice, there are both aesthetic and practical considerations that make it extremely difficult to do a really good algorithmic job. Mockingbird eventually will allow the user to modify the graphical representation directly, although it currently provides no such capability.

2.1.3.1 logical objects

Most of the objects in the logical domain have a representation in the graphical domain. (Voices are an exception, they are only implied in the graphical representation.) For instance, notes are represented in the graphical domain with a note head, stem and flag. The duration of
the note is determined by the shape of the note head and the number of flags present. As another example, beamed sets are indicated by replacing the flags of the notes with beams. The beams connect the beamed set's notes together. The duration of each note is now determined by the shape of the note head and the number of beams on the note.

2.1.3.2 staffs and clefs

A staff is the grid upon which pitches appear in a score. Pitch is on the vertical axis, time on the horizontal. A particular staff will cover only a small part of the range of possible pitches. This range can be extended up or down somewhat by using ledger lines. There are a number of possible origins for a staff; a clef is the symbol that indicates the origin of the staff within the range of pitches. In piano music there are two main clefs: bass and treble. Their origins can be moved up or down an octave by using octava notation.

A fairly broad section of the range of possible pitches can be covered by using a combination of several staffs. Each staff can be thought of as a window on the range. The windows may be disjoint or may overlap.

2.1.3.3 spacing and placement

A note's time of occurrence is given by its horizontal placement on the staff relative to the other notes. The absolute placement for each note is a property of the graphical representation. There are a number of considerations that affect the ultimate placement of an object: getting things as dense as possible, indicating the duration of a note by spacing, left and right justification of measures, and resolution of objects that interfere with one another graphically. Rather than give an exhaustive discussion of how publishers satisfy these requirements, we give examples of scores that show the problems and solutions. See figures 2.1.5 and 2.1.6.

2.1.3.4 stem direction and beam tilt

The stem direction and beam tilt are graphical aspects that are not determined by the logical representation of the piece. Publishers can use these to give hints about the structure of the piece, or to aid the performer in the mechanics of playing. For example, it there are several voices on one staff, stem directions might be used to distinguish between different voices. In another context, a beam's tilt might be used to give the general direction of a short run of notes. The performer can use this information to decide which finger the run should be begun with. Sometimes stem direction can influence beam placement, or vice versa.
2.2 MOCKINGBIRD'S MODEL OF MUSIC

We found that one of the most important decisions we made was how we should model music. After a great deal of thought, we came to the conclusion that a sequential model was the best one for representing music in an editor. Our goals for a model were that it be general enough to represent all three domains, flexible enough to allow the composer to say what he wanted, and forgiving enough to handle inconsistencies generated in the process of editing. The model should incorporate only those constraints that were absolutely necessary; we wanted to give the composer all the freedom we could. In the following sections we explain the sequential model of music and then discuss its advantages.

2.2.1 A Sequential Model of Music

We model music as an ordered sequence of events. An event is either a single note or a group of notes that occur at the same "time". The meaning of "time" is peculiar to each domain. In the physical domain, time means the number of seconds from the beginning of the piece. In the logical domain, it now means the number of beats from the beginning of the piece. Finally, in the graphical domain, it means the number of inches from the beginning of the score.

There are two rules that generally hold across all three of the domains we have discussed. The first rule is that the notes of an event are preserved as you pass from one domain to the next. Notes that are played at the same time in the physical domain usually occur on the same beat in the logical domain and are usually drawn aligned in the graphical domain. The second rule is that the events in each domain are ordered, and that the order is preserved as you move from one domain to the next. Thus notes that are played before one another in the physical domain usually occur before one another in the logical domain and are drawn before one another in the graphical domain.

These rules are violated by inaccurate playing or by unusual cases (rolled chords, grace notes, trills, chords containing adjacent pitches, etc.). The sequential view of music also ignores the fact that music may have repeats in it that tell the performer to go back to a prior part and begin there. In spite of this, we have found the rules to be quite useful for modeling music in Mockingbird.
2.2.2 The Advantages of a Sequential Data Structure

The two main advantages of a sequential data structure that we found while programming Mockingbird are that only one data structure is needed to represent all three domains and that the data structure behaves in a reasonable fashion while it is being edited. We also found that it was easier to write efficient procedures to manipulate the data structure.

2.2.2.1 one data structure for all three domains

The concept of a piece of music as a sequence of events captures important invariants that allow us to treat the different domains uniformly. The main difference between the domains is the choice of time. Other than this, all of the domains appear the same. As a result, a particular action can be implemented by just one procedure. For example, only one Replace procedure is needed to implement cuts and pastes in all three domains.

2.2.2.2 behaves well under inconsistencies

Often, music is in an inconsistent or erroneous state while being edited. This means that there is conflicting evidence about the state of the music (see example below). We feel that it is important for the editor to handle these inconsistencies gracefully whenever possible.

One source of inconsistency in music is the relationship between voices. There are two means of determining the time of a note: by its position relative to notes in its voice, and by its position relative to notes in other voices. These times may disagree. If they do, which should be relied on in presenting the note to the user? Our answer is that a note's time is determined by notes in the other voices. We adopt this answer because we believe that the concept of simultaneous events is so vital that it should not be violated unless the user explicitly says so.
Let's look at a simple example that shows the problem and our solution. Suppose the user enters the notes shown in figure 2.4. Voice 1 consists of a quarter note A followed by a quarter note B followed by a C. Voice 2 consists of a quarter note A followed by a C. Suppose also that the pairs of A's and C's were played together, as they appear. This leads to an inconsistency since it gives two different possible positions for the C₂ note: simultaneous with C₁ or simultaneous with B₁. C₂ might occur with C₁ since it was played with C₁, but note A's duration in voice 2 suggests that it should be with B₁. We assume that the user intended the first solution since that is how he played it. Our sequential model does the right thing and keeps C₁ under C₂ if they belong to the same event. Even if they do not belong to the same event, the relative order of the notes will be preserved unless the user explicitly changes it.

2.2.2.3 procedures are easy to code

We have found that the sequential model of music facilitates the writing of procedures. There are two reasons for this. First of all, notes are easily accessed. Many procedures are concerned only with the notes in a piece of music, and do not care about the higher levels of structure. In the sequential system, you need only to iterate over the events to obtain these notes.

Second, events are a useful data structure. Several procedures are easy to code if you can find out what notes are simultaneous. For example, you can write a simple procedure for playing music. If you know when any one note in an event is played, you know when all of the
notes in the event are played. Playing logical music is reduced to finding out when events occur in the logical domain.

2.2.2.4 time critical actions are sequential

It is a fortuitous result that those actions which take the most time are more efficient when the data structure is sequential. This is because the best algorithms for these actions are sequential. There are two time critical actions: drawing the score and laying out the notes. We discuss their algorithms in sections 5.3.2 and 5.3.3.

2.3 WHY NOT AN AUTOMATIC RECOGIZER?

Score “recognition” involves taking a particular pianoroll and trying to deduce its structure. In our terms, this means trying to infer the logical piece from a physical representation. This process is much like the process of understanding continuous speech. There is a prevalent belief that score recognition is relatively easy. This myth bears dispelling.

To demonstrate the difficulty of score recognition we will discuss just a few of the sub-problems: determining the durations of notes, deciding the voicing of a piece, and deciding how notes in a voice should be beamed. Each of these issues has problems that are very difficult to overcome, some of which even experts will disagree about. Taken together, they put score recognition in the same league as speech understanding.

2.3.1 Determining Note Durations

One of the most important things an automatic recognizer must do is determine the durations of the notes. In the physical representation, durations are measured in seconds: in the logical representation, they are measured in beats. A recognizer must find some way of correlating the two. We will first discuss the problem in a simple, idealized case and then proceed to show that as we add more realistic aspects, the problems rapidly multiply.

Suppose we start with a pianoroll that has only a single melodic line; no other voices and no chords. We already know the pitches, all we need to find are the durations. How might we do this?

The obvious first approach would be to look at the notes’ physical durations (how long the keys were held down). A reasonable assumption would appear to be that all quarter notes are held for about the same length of time, that quarter notes will be held for about twice as long as eighth notes, etc. However the fact that notes may be played staccato or legato (which shortens
or lengthens their physical duration without disturbing their logical duration) invalidates this simple assumption. Tied notes, which must be broken into their component parts, present a further problem as do the somewhat arbitrary decisions concerning how the composer might want to represent ornaments, grace notes, trills, etc. These considerations have led us to conclude that in general the physical duration of a note gives little if any clue to its logical duration.

If we look more closely at the meaning of the logical duration of a note, we find that its primary purpose is to define the logical distance (number of beats) to the next note in the voice. So instead of using physical durations to determine logical durations, let us explore what happens if we use the distance between notes. Our assumption now will be that all quarter notes have the same distances and that quarter notes have roughly twice the distances of eighth notes, etc. How well does this assumption work?

Again, there are problems. First of all, music is not performed at a constant tempo. The distance after a quarter note in one section of the piece may be as small as the distance after an eighth note in another section. The tempo of a piece may change because the composer really intends to make an explicit shift, or merely as an interpretive gesture that will not be explicitly shown in the score at all. If the composer decides that certain passages should be played at different speeds, he indicates this with marks such as allegro or andante. These shifts cause abrupt changes in the tempo. More gradual changes are achieved with the use of accelerando or ritardo. A tempo brings one abruptly back to the original tempo. The composer may also use a fermata mark to stop the music temporarily. Such changes in tempo may be hard to distinguish from changes in note values.

But not all tempo shifts are so definitely marked. As he plays, the composer makes small interpretive changes in the tempo: notes that fall on the main beat may be held longer for emphasis; there may be a hesitation before beginning a new phrase; beamed notes may be played as a group with short pauses at the beginning and end. As a sensitive performer, the composer introduces irresistible conscious and unconscious distortions of tempo. Adaptive algorithms have been made to work with simple music played carefully in time but fail to deal adequately with even modest complexity.

A still worse problem is that different musical structures can produce similar durations due to rests, tied notes, grace notes and tuplets. When is a half note to be written as a quarter note
tied to a quarter note? It may depend on the time signature or on the composer's whim. Rests measure how long a note is not to be played. How then do you distinguish a quarter note from an eighth note followed by an eighth rest, or a dotted eighth followed by a sixteenth rest? Grace notes present yet another problem. Logically they occupy no time at all, have no beats, and are not supposed to interrupt the flow of music. In performance, however, grace notes are played by borrowing time from a neighboring note. (The grace note starts a little before the next note, and/or the next note is delayed a bit. The normal tempo resumes with the note after it.) On the other hand, one composer may choose to spell out with "real" notes what another would spell with grace notes. Similarly for trills. Tuplets also cause difficulty because they allow notes to have non-binary durations. It may be quite difficult to distinguish the difference between a triplet of eighth notes, and two sixteenths followed by an eighth. It will be even more difficult to distinguish a seven-tuple of sixteenths from six sixteenths and an eighth. The fundamental problem is that we are trying to unmap a function that is ambiguous (see figure 2.6). There are often many ways to notate a particular section of music and the choice may be dependent on context or may be merely a matter of how the composer wants the passage to be viewed.

But all of these problems pale into insignificance by comparison with a more fundamental problem. Recall that we said that the logical duration of a note defines the distance to the next note of that voice. With a single voice, identifying the next note is straightforward; but with several voices, the problem suddenly becomes: which next note? The note you choose will determine the logical duration for this note. Unfortunately the next note in a voice may be some considerable distance downstream from the present note (or can even be a rest!). There can be an arbitrary number of intervening notes which belong to other voices. And, as we have seen, a voice may simply stop for awhile. What then? Certainly before we can even begin to attack all of the other problems, we must cope with the problem of separating a piece into its voices.

2.3.2 Separating a Piece into its Voices

Voices often cross one another. They disappear in places, only to reappear later. Even piano music, which is to be played by one person, is divided into voices. Sometimes a single note fits logically into more than one voice - and may have different logical durations in each.
Understanding how a piece of music is divided into voices requires not only an understanding of the harmonic and melodic structure of the music but also knowledge about completely arbitrary decisions that the composer must make concerning how he wants these structures presented in the score. For heavily rule-bound classical music, one can make some inroads into this problem; but for romantic and contemporary music, the rhythmic and harmonic complexities can lead into substantial arbitrariness about which the composer may nonetheless have extremely strong feelings.

We are therefore forced to conclude that if we want to deal with complex modern music, any attempt to deduce the voice structure algorithmically is not only extremely difficult but also certain to be inadequate and/or incorrect in many cases. This, in turn, means that we cannot infer note values with any certainty.

2.3.3 Grouping Notes with Beams and Phrase Marks

There are a number of objects that group notes horizontally: beams, phrase marks, and measure lines. There are two ways that an automatic recognizer might try to reconstruct this sort of information. The first would be to guess at the groupings by using the subtle emphasis that may be present in the physical performance. This emphasis would be in the form of pauses and louder notes. Unfortunately, these subtle analog variations tend to be masked by the larger variations in tempo and loudness. The second way to deduce such groupings is to understand the music structurally, just as in the problem of deducing voicing. In fact, beaming, phrasing, and voicing are intimately related to one another and are all vehicles by which the composer reveals (his notion of) the underlying musical structure of his composition to the reader.

Our conclusion is that trying to reconstruct the structures intended by the composer is a problem we did not want to tackle here.
1) grace notes  
2) notes shared between voices  
3) clef change  
4) new voice begins (or gap in voice)  
5) adjacent pitches in a chord  
6) adjacent chords in different voices  
7) rolled chords
Figure 2.2- Leo Ornstein, Piano Sonata #4, page 31

1) change in number of staffs (3 to 2)  
2) change in time signature (4/4 to 9/8)  
3) change in key signature  
4) 13-tuplet  
5) beam across measure boundary  
6) beam continued from previous line
Figure 2.3- Bach-Busoni, Chaconne in D minor, page 17

Three different ways to represent what is essentially the same musical structure.
Figure 2.5- Example Events
<table>
<thead>
<tr>
<th><strong>Composer’s Intention</strong></th>
<th>[\text{\textbullet\quad\textbullet\quad\textbullet\quad\textbullet\quad\textbullet\quad\textbullet\quad\textbullet}]</th>
</tr>
</thead>
<tbody>
<tr>
<td>**Pianoroll}_1) (staccato)</td>
<td>[\text{\textbullet\quad\textbullet\quad\textbullet\quad\textbullet\quad\textbullet\quad\textbullet\quad\textbullet}]</td>
</tr>
<tr>
<td>**Analysis}_1)</td>
<td>[\text{\textbullet\quad\textbullet\quad\textbullet\quad\textbullet\quad\textbullet\quad\textbullet\quad\textbullet}]</td>
</tr>
<tr>
<td>**Pianoroll}_2) (miss-timed)</td>
<td>[\text{\textbullet\quad\textbullet\quad\textbullet\quad\textbullet\quad\textbullet\quad\textbullet\quad\textbullet}]</td>
</tr>
<tr>
<td>**Analysis}_2)</td>
<td>[\text{\textbullet\quad\textbullet\quad\textbullet\quad\textbullet\quad\textbullet\quad\textbullet\quad\textbullet}]</td>
</tr>
<tr>
<td>**Analysis}_6)</td>
<td>[\text{\textbullet\quad\textbullet\quad\textbullet\quad\textbullet\quad\textbullet\quad\textbullet\quad\textbullet}]</td>
</tr>
</tbody>
</table>

*Figure 2.6- Problems with Automatic Recognition*
3.0 EXAMPLE

The following figures are snapshots of a Mockingbird session. Each figure title gives the list of commands that the user issued to get from the previous figure to the current one. At the bottom of the figure is some material that explains what is shown or the commands used.

The figures themselves were generated by printing bitmaps of the Dorado screen as the user used Mockingbird. The music you see is exactly the scale and resolution that the user sees as he edits. The bottom part of the screen has been clipped to leave room for the accompanying text.

Due to technical difficulties with the printing, the first twelve diagrams were generated by a different means than the last twenty-eight. We apologize for the poor copy that resulted.
A session with Mockingbird begins with an empty score. The view is preset to a four staff pianoroll (see section 4.1.2.2) using "in-the-cracks" notation for the pitches (see section 4.1.2.3). To record, the user selects a section of the score to record into and turns to the synthesizer to play.
Figure 3.2: *Record*

The user indicates to Mockingbird that he is through. Mockingbird replaces the user's selection with the pianoroll generated by his performance. The horizontal position of each note is directly proportional to the time at which the note was played. The length of each note is proportional to the duration of the note (how long the note's key was held down).
The user looks at the overview of the pianoroll.
Figure 3.4: Playback, Beat in measures

The user requests that the music be played back. There is a cursor (not shown in the figure) which tracks the music as it is played. While the music is playing, the user can strike the space bar once for each measure line he wants to insert. The measure lines are deposited a little before the cursor's position in the score.
Figure 3.5: look notehead

The view of the piece is changed so that logical durations are used instead of physical durations. Since no logical durations have been assigned, the notes appear as X’s.
Figure 3.6: look sync

The view of the piece is changed so that sync lines appear on the score. These sync lines show the notes that were played almost simultaneously when played in. Notice that the first notes in the sixth and the ninth measures have not been included in the nearby events because they fell outside of Mockingbird's tolerance for "simultaneity". The user can correct this problem manually, or he can use a more tolerant syncing heuristic.
Figure 3.7: everything

The user issues the command "everything" which makes a section selection of the entire score.
Figure 3.8: automatic sync

The user issues the command "automatic sync". Mockingbird syncs together all notes that are within a short time of one another. This algorithm is more liberal than the algorithm Mockingbird used when the pianoroll was read in, and so a few more notes are synced together.
Figure 3.9: *ESC*

The user types ESC to clear the current selection. From now on we will be less meticulous in showing how selections are used.
Figure 3.10: *look non-sync*

The user changes the view so that sync lines no longer appear on the score.
Figure 3.11: (select notes)

The user selects all of the notes that are to go into voice 1 by pressing down the red button and sliding the mouse over the notes. If a note is erroneously selected, the user can deselect just that note by clicking the yellow button over it. Selected notes are shown in grey.
Figure 3.12: *voice 1, look voice 1*

After putting the selected notes in voice 1, the user changes the view so that only notes in voice 1 appear. Any subsequent actions can only affect these notes as long as this view remains.
Figure 3.13: look sheet 2

The user changes the view so that the piece appears on a two staff sheet. Note that the notes have been arbitrarily split at middle C: all of the notes at middle C or above appear on the first staff, and all of the notes below middle C appear on the second staff.
Figure 3.14: everything, staff 2

The user puts all of the notes on the second staff.
Figure 3.15: *everything, eighths, automatic chording*

The user sets all of the notes' logical durations to "eighths". The notes are not supposed to be all eighths, but this is a good first approximation. The user then issues the command "chord" which is interpreted as "automatic chord" since a section selection was made. This causes Mockingbird to chord notes which have the same event.
Figure 3.16: (note select), quarters

The user goes back and selects a number of notes which should have been quarters rather than eighths. After the notes have been selected, he issues the command "quarters" once and the logical durations of all of the selected notes are set to quarters.
Figure 3.17: (note selection), dotted quarters

The same process is repeated for the notes which should have been dotted quarters.
Figure 3.18: (note selection), dotted eighths

There is also a note which should have been a dotted eighth.
And finally there are a few notes which should have been sixteenths. Now the user wonders if he got all of the durations right. If the piece had a time signature, Mockingbird could help the user check the durations by making sure that the voice "adds up" within each measure.
Figure 3.20: *everything, time signature: 6/8*

All of the measures now have a time signature of 6/8. (Time signatures can be set on a sectional basis if need be).
Figure 3.21: constraint checking

Now that the piece has a time signature, the user asks Mockingbird to check the measures to see if the durations add up correctly. Mockingbird finds four bad measures. The sum of the durations in each of these measures is not the same as the time signature would indicate.
Figure 3.22: *insert note, (note selection), set durations to 32nds, stems up, insert rests*

The user fixes up all of the mistakes by changing note durations and inserting the appropriate notes and rests. He also turns the 32nd notes’ stems up for clarity. The note added to the first measure is necessary to complete the measure and is in fact a duplicate of a note in the other voice. This duplication is manifested in the published version by a double stem. (See figures 3.40 and 3.41.)
Figure 3.23: constraint checking

The user checks the durations again to make sure that all of the mistakes were corrected. This time, Mockingbird only complains about the first measure. (First measures are often incomplete.)
Figure 3.24: (note selection), beam

The user selects a number of notes and issues the command "beam". The notes then appear connected together by a horizontal beam positioned an appropriate distance from the note heads.
Figure 3.25: (note selection), beam

The action is repeated for a new set of notes. Beaming can be done fairly quickly since the notes can often be selected with one mouse movement.
Figure 3.26: (*note selection), beam

This beam shows that Mockingbird can handle beams that wrap around the edge of the page, and that it can display beamed sets that contain more than one type of duration (the beamed set is made up of an eighth, a dotted eighth, and a sixteenth).
Figure 3.27: (note selection), beam
Figure 3.28: *(note selection), beam*

The last beam is made up of two thirtysecond notes.
Figure 3.29: *look voice 0*

All of the notes that were not put in voice 1 by the user are in voice 0 by default. When the user sets the view to voice 0, these notes appear and the notes from voice 1 disappear. All subsequent actions only affect the notes in voice 0.
Figure 3.30: *everything, staff 1*

The user puts all of the notes on staff 1.
Figure 3.31: *everything, eighths, automatic chording*

The user begins to assign durations to the notes in voice 0 much as he did in voice 1. All of the durations are set to eighths (even though they should not all be eighths) and all of the notes with the same event are chorded together.
Figure 3.32: (note selection), quarters, (note selection), dotted halves, (note selection),
dotted quarters

As before, the user goes back and assigns the correct durations to the notes.
The user asks Mockingbird to check the note durations. Mockingbird is satisfied with all of the measures except the first (which has no notes) and eighth (which is missing an eighth rest).
Figure 3.34: *insert rest, constraint checking*

The user corrects the mistake and checks the score again. He is now satisfied.
Figure 3.35: (note selection), beam, (note selection), beam, (note selection), beam

The user beams several groups of notes together into beamed sets. Note that chords can be beamed together as well as notes.
Figure 3.36: *look all voices*

Having finished working with the two voices separately, the user changes the view so that he can work on them together.
This time Mockingbird takes into account the interactions between the two voices. It does not like the fourth measure because the second thirty second note is erroneously synced with the dotted quarter note above (see Figure 3.8). The duration of the measure is thus off by a thirty-second. Similarly, Mockingbird does not like the eighth measure because the rests in the bottom voice are not properly synced with the notes in the top voice.
The user ignores these problems and asks Mockingbird to justify the measures. The justifier aligns the notes, spaces them aesthetically, and justifies the measures. This process moves the piece from the physical domain into the logical domain. (The alignment problems in the fourth and eighth measures were fixed by the justifer. See section 4.1.4.3)
In previous views, the user chose to use the "in-the-cracks" notation (see section 4.1.2.3). He could have set the key signature at any time, but the accidentals would have cluttered the display. The justifier avoids this clutter by making room for the accidentals. In this figure, the user has finally defined the key and then cleaned up the spacing with a second justification.
Figure 3.40: tilt beams, (note selection), staff 1

The user tilts some of the beams to make the score look prettier. He also moves the first note of voice 1 into staff 1 to make it appear that there is one note with two stems.

This is as far as Mockingbird goes currently. To finish this piece, the user would have to be able to tie notes, make notes mordents, draw phrase marks, and make comments.
à Mlle de Noailles

Troisième Ballade

Revised, edited and fingered by Rafael Joseffy

F. Chopin. Op. 47

Figure 3.41- Chopin, Third Ballade, first nine measures
4.0 THE USER’S VIEW

This section deals with Mockingbird as it appears to the user. It is broken into two parts: user actions and the user interface. The first deals with the editor’s functionality: those actions that the user can take to modify the data structure or its view. The second is concerned with the forms of these actions: the particular keystrokes or mouse movements that will invoke a user action. This section discusses all the user needs to know to be able to use the editor. Before we discuss the actual interface, we will cover the goals we had for Mockingbird and the metaphors that influenced its design.

4.0.1 Goals

Mockingbird was designed with several goals in mind. First of all, we wanted an editor that was quick. A music editor should have fast means for both capturing and modifying music. Of the two, we felt that quick capture was the more critical to avoid losing ideas. Once an idea has been captured, the speed of response becomes more a matter of convenience than of necessity.

Next, we wanted an editor that was flexible. It should not have too many preconceptions about music or about the editing process. We did not want to forbid certain styles of music, nor did we want to constrain the user to a particular order of editing. The editor should help the composer without getting in his way.

Finally, we wanted an editor that was simple. Musicians may not be computer scientists, or even "mechanically" oriented. The metaphors involved should be transparent, consistent and intuitive; transparent in that a particular action is easy to understand, consistent in that the set of actions does not use conflicting metaphors, and intuitive in that the actions are easy to remember.

4.0.2 Metaphors

Mockingbird draws its metaphors from the two domains of music and of text editors. Music and text have much in common, since music is a sequence of events, much as text is a sequence of characters. Most of the actions in Mockingbird’s editor correspond to activities a composer normally performs. Thus the editor includes the concepts of "beaming" and "chording" notes. Some of the actions correspond to functions found in text editors. For
instance, the editor has the concept of cutting and pasting sections of a data structure. This commonality makes it easier for users familiar with either music or editors to grasp the essence of the interface quickly and easily.

4.0.2.1 **functionality**

Like text editors, music editors need a way to select and manipulate their sequences, and to change the value of attributes. The two types of editors also share the forms of actions. Replacement is the most general form of a sequence manipulation. Replacement takes as parameters two selections: primary and secondary. The primary selection is replaced with a copy of the secondary selection. Deletion and insertion are special cases of replacement. If the primary selection is empty, the command becomes an insertion; when the secondary selection is empty, the command becomes a deletion.

The atomic elements of a music editor and a text editor are notes and characters respectively. Both notes and characters have attributes that can be modified by the user. The character's attributes are its font, its boldness and its italicness. The note's attributes are its pitch, time, value, voice, staff, and stem direction.

This is as far as the similarity goes. A musician must be able to beam, chord and sync notes together, as well as insert, delete and replace. These actions have no correspondence in text, although they fit well into the "selection-command" form discussed below.

4.0.2.2 **form**

In Mockingbird, actions have a common form: part of the music is selected, and then an action is taken that affects those items selected. This "selection-command" form is shared by both types of editors, and is widely used. Mockingbird allows the user to select a number of items before issuing a command. The user may then issue as many commands as he pleases on this collection of selected items. The cycle can then be repeated with new selections and new commands.

Not all of the music editor's commands fit into this form. The most recalcitrant commands involve the insertion and movement of notes. Since these commands do not fit into our standard mold, we discuss them separately in section 4.2.4.

4.1 **USER ACTIONS**

The user actions can be grouped by function into four sets: utilities, view manipulations,
data manipulations and miscellaneous. The utilities coordinate the editor with its environment. They allow the user to store and retrieve music as files in the Dorado file system, play and record on the synthesizer, and return control to the Dorado executive when the user is done. View manipulations change how the music is perceived by the user. The view is the sum total of all the parameters that affect how the data structure is displayed or played. Data manipulations change the music itself. Miscellaneous actions encompass heuristics, constraint checking, and layout.

4.1.1 Utilities

There are five utility actions in Mockingbird: Record, Play, File In, File Out, and Quit. The first two, Record and Play, coordinate the editor with the synthesizer. The second two, File In and File Out, coordinate the editor with the Dorado file system. The last action, Quit, coordinates the editor with the Dorado executive.

4.1.1.1 recording

Recording is the act of transforming keystrokes on the synthesizer keyboard into a pianoroll. This action is treated as a replacement. The user selects a section of the music which is to be replaced by his performance. (Empty selections are acceptable.) He then turns to the keyboard and starts playing. The computer waits for his first keystroke before recording. When the user is finished, he turns back to the editor and indicates that he is through. The editor then takes the pianoroll just entered and splices it into the music where the user had indicated.

4.1.1.2 playing

Playback is an audible "view" of the music. This action converts the editor's internal data structure into a sequence of keystrokes. These are then given to the synthesizer, which produces sound. Playback allows the user to listen to the music that he observes on the screen. The computer's performance is unswerving and thus sounds somewhat mechanical, but it is sufficient for the composer to proofread what he has produced.

Playing a piece of music may involve switching domains on the fly since the score may be a quiltwork of physical and logical sections. Mockingbird detects the boundaries between different sections and handles the transitions smoothly. We explain how this is done in section 5.4.2.

The editor tracks the music being played with a cursor, indicating exactly where it is in the score. While the editor is playing a piece of music, the user can play against himself or beat
measures in. To do the first, the user simply turns to the synthesizer keyboard and starts playing. Whatever he plays will be merged into the score at the appropriate place. To beat measures in, the user strikes the space bar on the beats of the music. A measure line is inserted a little before the position of the cursor when he hits the space bar (This makes an allowance for the human response time).

4.1.1.3 filing in and out

Music can be stored on and retrieved from external files; the editor has two actions, File In and File Out, for this purpose. Mockingbird uses a special file format that contains not only the data structure but also view information. Filing in a file sets the editor's view to that which it had when that piece of music was filed out.

4.1.1.4 quitting

When the user is finished, he can terminate the editing session by quitting. All of the editor's state is lost when the user quits. To prevent accidental loss, quitting, filing in and filing out are protected actions. The editor knows when a file has been dirtied and will not allow it to be destroyed unless the user confirms the action.

4.1.2 View Manipulations

The editor's view of music can be manipulated by the user. The view is the sum total of all the parameters that affect how the data structure is displayed and played. It determines what is on the screen and how it appears to the user. The window determines the what; it is the portion of the screen dedicated to displaying the data structure. The rest of the parameters are concerned with the how. (We think of the latter parameters as filters on the data structure.) We chose the aspects of the view so that they are independent; changing one aspect does not affect another. This means that we can discuss the different aspects separately.

4.1.2.1 scrolling and scaling the window

The Dorado screen is of limited size so that some means are needed to allow the user to look at different parts of the data structure. To accomplish this, Mockingbird lets the user scroll and scale the window. Scrolling moves the window backwards or forwards through the data structure. Scaling changes the amount of music viewed by the user.

Mockingbird has two display scales; normal and condensed. In the latter scale, the user can see sixteen times as much music. This allows the user to get an overview of the music; he
can find things easily and do gross editing operations although detail is lost. In contrast, the user cannot see as much of the score in the smaller scale, but it is easier to perform fine editing operations.

4.1.2.2 choosing the sheet

The sheet is the template upon which the data structure is displayed. The number and type of staves, the distance between the staves, and the distance between the lines are all determined by the choice of sheet. There are several sheet types to choose from:

1) two staff sheet - a treble and a bass,
2) three staff sheet - a treble and two bass,
3) three staff sheet - two trebles and a bass,
4) four staff sheet - two trebles and two bass,
5) four staff sheet - a treble, a bass, a staff two octaves above treble and a staff two octaves below the bass.

In the first four types, the inter-stave spacing is just like regular music score sheets. However, the last sheet type is used for displaying pianorolls and on it the distances between the staves are chosen so that pitch is linear along the vertical axis. The staves cover the entire keyboard without duplication or octava notation. The result is that there is a unique correspondence between pitch and note position on the staff.

4.1.2.3 choosing the key

Another aspect of the view is the choice of key. All of the major and minor keys are available to the user. The user may specify the key that the music is to be displayed in on a sectional basis. This allows the user to change keys in the middle of a piece. The key determines how a certain pitch is to be displayed; what its accidental is to be. Mockingbird uses a simplistic rule for determining accidentals based on the key: if the key has flats in its signature then the accidentals will all be flats, if the key has sharps in its signature then the accidentals will all be sharps. The usual rules for carrying accidentals forward to the end of measures (or next accidental on that line) are also enforced. Currently the user has no control over what accidental will be used for a particular pitch.

Alternatively, the user may decide that there is to be no key. This is not the same as saying the default is the key of C; when there is no key then no accidentals are displayed at all. In this
situation the editor displays the black keys "in the cracks"; black keys' notes are positioned between the line and the space (See figure 3.2). It is often convenient to display pianorolls in this manner so that there are no accidentals to obscure the notes.

4.1.2.4 showing the voices

Another aspect of view is the choice of voices. The user may suppress some voices in the view to make another voice clear, or to be able to issue commands that affect a large section of the voice. Normally, all of the voices are displayed. This allows the user to see their interactions. The user can change the view so that only the voice he is concerned with appears on the screen. The voice can be edited, tested, and even played by itself. For example, the user may select some voice and set all of its stems up without affecting any of the notes in the other voices.

The ability to filter the data structure by voice raises a number of questions about selections. What happens to the selection when you change voices? Do the selected notes in the other voices remain selected even though they are no longer seen on the screen? Can you select notes that are not in the current voice? Do actions affect notes selected in other voices? These questions boil down to one question: which has precedence, the selection or the voice?

We answered this question by formulating a principle: The user should not be able to change what he cannot see (the user can "see" something if it is visible on the screen when he scrolls to the appropriate section). We see the selection as applying to the data structure, and then the voice filter is applied to the selection and data structure together. The user can select or act on only those notes that are visible. Notes are not deselected when the view's voice changes, they become invisible. Since actions affect only what is visible, they cannot affect selected notes that belong to a different voice.

4.1.2.5 showing the event lines

The user can choose to show the "event lines". These are vertical lines that indicate which notes are grouped together into an event. They distinguish between notes that are merely close together in time and notes that are defined to be simultaneous. Generally, the event lines are of concern at the beginning of the editing process when the user is establishing and verifying proper synchrony. Later on they only clutter up the display so the user will want to suppress them.
4.1.2.6 **showing the durations**

There are two options for displaying the duration of a note; physical and logical. The duration of a note is the length of time that a note is held down. Physical durations are measured in seconds, logical durations in beats. These two durations have two different representations on the screen. A physical duration is shown as a black bar, where the length of the bar indicates the duration of the note. This display mode is useful for pianorolls. A logical duration is shown as one of the standard music notation note heads (whole note, half note, quarter note, etc.) along with the appropriate flag. If no logical duration has been assigned, the note head will appear as an X.

4.1.2.7 **playing the music**

Playing music is regarded as a "view" affecting the ears rather than the eyes. The user can modify this "view" by changing the way the editor performs a piece. Currently, the only aspects of the audible view which can be modified are the speed at which the music is played and the choice of voice. The user has access to the internal metronome that the computer uses to time its playing. By setting this metronome, he can choose the speed at which the music is played.

4.1.2.8 **choosing the domain**

The user can choose the representation used by the editor to display a piece. The choice of representation will affect how the notes are spaced. If the user chooses the physical domain, then the editor will display a physical representation for the piece. Notes will be placed on the screen at a position that is linearly related to their times of occurrences. This spaces notes in the same way notes in a pianoroll are spaced. If the user chooses the logical domain, then the editor will display a logical representation for the piece. Notes are placed on the score with inter-note distances that are related logarithmically to their duration. This spaces notes somewhat as notes in a published score are spaced.

4.1.3 **Data Manipulations**

The following user actions change the data structure itself. The actions can be grouped according to the part of the data structure that they change. Some actions focus on the sequence of events. (They see the music as a sequence and proceed to rearrange this sequence.) These actions manipulate the data structure in a very coarse manner. At a finer level of detail, individual notes may be added, subtracted or moved. Other actions change the attributes of a
The last group of actions is responsible for grouping notes.

4.1.3.1 manipulating the sequence of events

At the grossest level, the user can manipulate the sequence of events with Replace, as discussed before. Replacing has two different meanings, depending on the context. If all of the voices are being displayed, then a cut will destroy the underlying sheet and the section’s neighbors will abut one another. If any voice is omitted from the display, then a cut does not affect the underlying sheet although the voice that was being displayed will be deleted as if it had been erased from a blackboard. We do this to be true to the principle mentioned above: the user cannot change what he cannot see. We erase the voice rather than destroying the sections since other voices may be on the sheet at this point, and destroying the sheet would affect those voices.

Recording from the synthesizer is treated as replacing, only instead of taking music from another part of the piece, the editor takes in what is played. The notes recorded from the keyboard are stored as a pianoroll.

4.1.3.2 adding or subtracting notes

At a finer level of detail, individual notes or objects may be added and subtracted. These actions do not affect the placement of other notes. Notes are added to or subtracted from a particular event. This only affects the sequence of events if a new event is created or the last note of an event is removed. Measure lines are treated differently from other objects. They are considered to be events in themselves so that inserting or deleting a measure line changes the sequence of events.

4.1.3.3 grouping notes into events, chords, beams, and tuplets

Each of these actions builds a special type of data object. A particular note can belong to at most one element of each type. The actions are designed to enforce this constraint by removing the note from its old object before it is added to the new one. This means, for instance, that if a note is shared between two chords, there must actually be two instances of the note, one for each chord. Such double instances will typically overlie one another and thus will look on the display like a single note.

Syncing collects notes into a single event. The notes are first removed from their former events and then put into a newly created event. This event is then added to the sequence of
events. The converse action for syncing is de-syncing. De-syncing breaks an event down into its component notes. Each note becomes its own event (the order of these events does not matter).

Chording collects notes into a single chord. When notes are chorded, they are first removed from their former chords, if any. They are then formed into a new chord. De-chording, the converse action, destroys the chord structure and allows the notes to become free entities. An additional constraint we impose on chords is that all of the notes in a chord must belong to the same event. We enforce this constraint by moving the notes from their event to the chord’s event when the notes are chorded. (The time of the chord’s event is given by the time of the first note selected for inclusion in the chord.)

Beaming collects notes and chords together into a horizontal structure. The notes are first removed from their former beam structure (if any) and then bound up into a new beam structure. De-beaming destroys a beam structure.

Tuplets are a special class of beams. There are two ways for the user to make a tuplet: by changing an existing beam into a tuplet, or by creating a tuplet from scratch (in the same way that a beam is created). These produce two different types of tuplets. The difference is in their appearance: the first groups the notes together with beams, the latter with phrase marks. We have both types of tuplets since traditional music uses both.

4.1.3.4 changing note attributes

There are a number of actions that change the attributes of notes. Currently we recognize four note attributes: value, voice, staff, and stem direction. Later we plan to add performance attributes (staccato, legato, loudness, embellishments) and graphical attributes (positioning of accidental, notehead, and dot).

The value of a note is the fractional number of beats that the note occupies. A note may be whole, half, quarter, eighth, sixteenth, thirty-second, or dotted versions of these.

The voice of a note indicates the musical conversation of which it is a member. Voices may be identified by any of the numbers between zero and nine.

The staff number of a note indicates the staff on which it is to be displayed. This attribute is convenient for keeping the display uncluttered. The user may separate objects that conflict graphically by putting them on different staffs.

Stem direction is a completely graphical attribute. It determines whether the stem of the
note (or chord) is to go up or down and is often used to imply the voicing that the composer intends.

4.1.3.5 changing event attributes

There are certain attributes that apply to a section of music, rather than just to a note or chord. These sectional attributes are generally associated with an event. Therefore if the events get moved around, the modifications move with them. The only attributes of this sort that we have implemented so far are the key and time signatures. (Likely possibilities in the future are tempo and loudness markings). The user may change the key signature anywhere, and the time signature at any measure line. These attributes are changed by selecting a section of music and specifying the new attribute. Any key or time signatures in the selection will be removed. The new signature is in effect until the next signature beyond the selection.

4.1.4 Miscellaneous

4.1.4.1 heuristic actions

All of the above actions have explicit, algorithmic effects. The user always knows clearly what effects his action will have. These actions give the user the functionality he needs but not the speed he desires. To help, we provide a class of heuristic actions that speed up the composer's work. These actions do not always do exactly what the composer wants. Their advantage is that they are fast. All of our heuristics are optional; the user invokes them only if he chooses. If a user does not like our heuristics, he can ignore them.

So far, two heuristics have been implemented: automatic syncing and automatic chording. The first suggests sets of events for the notes. The second guesses at how the notes should be chorded. Later we will attempt more sophisticated heuristics which we hope will also improve the user's efficiency.

Automatic syncing tries to guess which of the notes in a pianoroll are to be considered simultaneous. Our rule is to group notes that are within a certain time-window into events. This rule is usually invoked after music has been recorded from the synthesizer's keyboard. The assumption is that notes that were played within a short time of one another were probably intended to be played together. This is true if the performer was careful and did not play any embellishments. If the performer was not careful, then the heuristic may sync notes that the user did not want so grouped.
Automatic chording tries to guess which notes should be grouped together into chords. Our heuristic is to chord notes that are both in the same event and in the same voice. Being in the same event means that they are simultaneous, being in the same voice means that they are related. Experience has shown us that this is usually close to what the composer intended.

4.1.4.2 constraint checking

In addition to providing tools for manipulating the view and data structure, Mockingbird provides constraint checking on the logical time of a piece. Each measure in a piece of music has a time signature that indicates how many beats of music are to be found in the measure. There is nothing inherent in Mockingbird that requires the measure to have this number of beats. The user may, however, ask Mockingbird to check it for him.

The constraint checker compares the longest path along the voices in each measure with its time signature and complains if there is a discrepancy by greying that measure. The longest path is determined by scanning the events, taking the time for each event to be the largest of the times of the component voices. The time of each voice is the sum of the last duration in that voice and the time of that duration's event. This incorporates the interactions between the voices.

4.1.4.3 layout

Layout is responsible for three things: voice alignment, (horizontal) note placement, and measure justification. Voice alignment rearranges the notes in a set of voices so that the notes are in their correct places relative to one another. Events are broken and regrouped to accomplish this. Two voices are only aligned by the program if each "fills" the measure. A measure is filled by a voice if the length of that voice matches the time signature exactly.

Note placement and measure justification are graphical operations. Together, they produce a score that approximates a publisher's placement of notes. The measure lines are left and right justified and the notes are placed in a manner that conserves space and is pleasing to the eye.

4.2 THE USER INTERFACE

In the prior section we discussed the actions that Mockingbird provides. In this section we discuss how those actions are invoked, and the forms of the responses. In addition, we talk about what motivated the design of the interface.

There are two types of commands: those that fit in the "selection-command" metaphor
discussed above, and graphical commands (movement and insertion). In the following sections we will discuss how selections are made, how "selection-command" commands are invoked, and how graphical commands are invoked. Before we do this, however, we will review the components of the interface.

4.2.1 Basic Tools

The interface is constructed using the basic components of the Dorado and a synthesizer. The Dorado's components include a keyboard, a mouse, and a full screen bitmap with cursor. The synthesizer has a 76 key keyboard and an array of sound producing elements.

4.2.1.1 screen

The screen is an eight-and-one-half by eleven inch raster scan display with a resolution of about 76 bits per inch. Each bit on the display can be individually turned on or off. By turning the bits on in a particular pattern, we can cause images to appear on the screen.

4.2.1.2 keyboard

The keyboard is a standard alphanumerical keyboard with some additional special characters. It also has a shift key and control key. Both of these keys are used to modify the other keys. Thus the key 'a' may be seen by the computer as 'a', 'A' (shift a), 'at' (control a), or 'A+ ' (control shift a).

4.2.1.3 mouse

The mouse is a special pointing device with three buttons on top. Different points on the screen can be indicated by moving the mouse around on the desk. The buttons are an I/O device; the programmer can use them for whatever he wants. They are usually labeled "red", "yellow", and "blue" for identification. Each button has two states: up and down. Usually, programs recognize three button actions: button pressed, button released, and button clicked (down then up). Which of these two states and three actions the program watches for depends on the command and the context.

4.2.1.4 cursor

The screen has a special cursor that can be associated with the mouse. The cursor is a 16x16 bitmap that is programmer definable. When it is associated with the mouse, the upper left corner of the cursor is on top of the point that the mouse is pointing at. Changing the shape of the cursor can give the user feedback from the editor.
4.2.2 Selection

Mockingbird has two types of selection; note selection and section selection. There are two types of section selection as well: primary and secondary. (Secondary selection is used only in replacement.) Note selection allows one or more individual notes to be selected; section selection allows a sequence of events to be selected. These two selection modes are mutually exclusive; only one type of selection may be in use at a time.

All of the selections are made with the red button on the mouse, modified by the control and shift keys: Note selection is done by the red button with no modification. Section selection is done by control red button. The secondary section selection is done by doing a shift control red button. Thus, all of the selection actions are grouped together for the user's convenience.

4.2.2.1 note selection

Any number of notes may be individually selected or deselected. When selected, a note appears grey. A note is selected whenever the mouse is over it and the red button is down. Thus the user may select a number of notes quickly by holding down the red button and sliding the mouse across them. The user may deselect all of the selected notes by pressing "ESC" on the keyboard. Individual notes are deselected by clicking the "yellow" button over them. One note is deselected per click. While the yellow button is down, the nearest note is flashed to indicate what the editor thinks the user is pointing at. The deselection is made upon release. This arrangement allows the user to select notes quickly with occasional mistakes and deselect notes carefully.

4.2.2.2 section selection

A section selection is a selection of a sequence of events. To make a selection, the user depresses the red button, slides the mouse along the section of music he wants selected, and then releases it. The beginning of the selection is the point where the button is depressed, the end is where the button is released. The section from the beginning to the current position is flashed as long as the button is down. When the button is finally released, the section is inverted to indicate selection. All of the music between the two endpoints is considered selected.

There are two types of section selection: primary and secondary. Only one selection of each type may be made at a time. The primary selection is invoked by control red button, the secondary by control shift red button. The primary selection is displayed by XOR'ing the
selection with black, the secondary uses grey. Both types of section selection may be in use at
the same time, and the selections may overlap.

4.2.3 Commands

There are two main ways to invoke actions: keyboard and mouse. Most commands are
entered via the keyboard; a small number are invoked solely by use of the mouse. The choice is
based on the type of action: if an action applies only to the current selection, then it is a
keyboard command. If an action requires that a position on the screen be indicated, then it is a
mouse command. Some actions have been made mouse commands in order to save room on
the keyboard. These are the utility actions, which are used infrequently. See figure 4.2 for the
list of commands.

4.2.3.1 keyboard commands

Most keyboard commands are invoked by striking a single character, some need two
characters, and some require parameters. Keyboard commands usually act on the current
selection. Generally, if a command p has a converse it is invoked by \textit{shift} p. For example, notes
are chorded with a c and de-chorded with a C. A keyboard command may be sensitive to the
type of selection used. For example, if individual notes are selected, then c causes them to be
chorded together; if a section is selected, then c invokes the editor's chording heuristic. The
commands' syntax is also given in figure 4.2.

4.2.3.2 mouse commands

The second way to invoke actions is by the mouse buttons. Some areas of the screen are
sensitive to certain mouse buttons. If a button is clicked, then an action is executed. There are
two sensitive areas on the screen: a "scrollbar" at the left and a "command menu" at the top.
Each of these areas has its own behavior. In general when a mouse button is pressed, the cursor
changes shape to indicate what action is going to be executed. The action is not invoked until
the button is released. Moving out of the sensitive area before the button is released will abort
the command.

The scrollbar at the left is used to change the window's aspects. The user can scroll up (red
button), change scales (yellow button), or scroll down (blue button) by appropriate actions with
the mouse. The distance scrolled up or down depends on the mouse's position on the bar. A
change of scale can be done anywhere in the bar.
The command menu at the top allows the user to invoke the utility commands. The commands were put there because we had run out of characters for keyboard commands. The user can file in, file out, playback, record, or quit from the menu at the top of the screen. The menu consists of a list of these command names. When the mouse passes over a command, the cursor takes on the appearance of a bullseye. Pressing any mouse button will then invoke the command.

4.2.4 Graphical Commands

There are two commands which do not fit into the "selection-command" form discussed above: moving and inserting notes. These commands are taken up in this section.

4.2.4.1 moving notes

Any note on the screen can be moved both horizontally and vertically. Notes that are to be moved are first picked up by pressing the blue button modified by a shift while the cursor is over the note. Thereafter, as the mouse moves around, the note follows. When the user releases the mouse button, the note is deposited at that point.

Vertical motion is not the same as horizontal motion. Moving the mouse to the left or right changes the note’s time (position on the score). The change in the note’s time is directly proportional to the change in its horizontal position. Moving the mouse up or down changes the note’s pitch (motion is discontinuous as the note moves among the discrete pitch positions). Because the pitches are graphically close together, we made large changes in mouse position cause small changes in pitch. The editor generates the correct accidental for the note depending on where it is placed. As a note is moved up and down, new accidentals are drawn in front of it according to the key and the nearby notes. It is very easy for the user to see exactly what the new pitch of the note will be. If the note belongs to a chord, the whole chord follows the horizontal position as the note is moved, but the only pitch that changes is that of the selected note. If the user does not want the chord moved as well, he should first remove the note from the chord. Finally, if a note (or the chord) is deposited very near an event line, it is automatically attached to that event. This saves a lot of manual syncing but it can occasionally cause notes to become parts of the wrong events. The user can undo this by explicitly resyncing the notes.
4.2.4.2 *inserting notes*

So far, the only method we have described for entering music is through the synthesizer keyboard. There is a second means for adding objects to the data structure that does not use the synthesizer keyboard at all. This means is called insertion and uses a special "pop-up" menu.

Insertion is a special case of note movement. Notes are normally picked up from somewhere on the screen and moved to another point. When inserting notes, the note is picked up from the special menu and moved to the appropriate place. Once a note type has been obtained from the menu, the user may deposit as many copies as he chooses. One way to think of this is that the cursor contains an infinite supply of that note.

4.2.4.3 *the special menu*

Mockingbird has a pop-up menu that contains icons of all of the objects a user might add to the data structure. Whenever the user wants to see this menu, he presses the blue button while holding down the control key. The menu pops up at the mouse's current position, allowing the user to choose an icon by moving the cursor around within the menu while continuing to hold the blue button down. As the cursor passes over an icon, it changes shape to match that icon. This way the user and editor can both know which icon is being chosen. When the mouse button is released, the editor remembers the icon. The cursor retains the icon so that the user knows that he is in one of the insert modes and what type of item will be inserted. Now, wherever the user points, he can deposit an object of the type "icon" by clicking the blue button. He can put down as many objects as he desires, clicking once for each object.
Figure 4.1- Mockingbird as it appears on the screen (reduced picture)
### Mouse Actions

<table>
<thead>
<tr>
<th>Context</th>
<th>RedBug</th>
<th>YellowBug</th>
<th>BlueBug</th>
</tr>
</thead>
<tbody>
<tr>
<td>screen command execute</td>
<td>...</td>
<td>overview</td>
<td>...</td>
</tr>
<tr>
<td>scroll bar scroll up</td>
<td>...</td>
<td>jump to detail</td>
<td>move beam (left=y, right=tilt)</td>
</tr>
<tr>
<td>overview select note</td>
<td>...</td>
<td>delete note/</td>
<td>move measure line/</td>
</tr>
<tr>
<td>(in score)</td>
<td></td>
<td>delete measure line</td>
<td>insert from menu</td>
</tr>
</tbody>
</table>

#### Control

<table>
<thead>
<tr>
<th>CONTROL-SHIFT</th>
<th>RedBug</th>
<th>YellowBug</th>
<th>BlueBug</th>
</tr>
</thead>
<tbody>
<tr>
<td>section select</td>
<td>...</td>
<td>...</td>
<td>select from menu</td>
</tr>
<tr>
<td>grey section select</td>
<td>...</td>
<td>dotted select from menu</td>
<td>move note</td>
</tr>
<tr>
<td>extend note select</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

### Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Any Selection</th>
<th>Section Selection</th>
<th>Note Selection</th>
</tr>
</thead>
<tbody>
<tr>
<td>Beam</td>
<td>...</td>
<td>heuristic beam</td>
<td>beam notes</td>
</tr>
<tr>
<td>Clear beams</td>
<td>...</td>
<td>heuristic chord</td>
<td>chord notes</td>
</tr>
<tr>
<td>Clear chords</td>
<td>...</td>
<td>delete section</td>
<td>delete notes</td>
</tr>
<tr>
<td>Set metronome to n (128 is standard)</td>
<td>...</td>
<td>section select + grey selection</td>
<td></td>
</tr>
<tr>
<td>Set notes to n/m n-tuplet (3/2=triplet)</td>
<td>...</td>
<td>section selection</td>
<td>sync notes</td>
</tr>
<tr>
<td>See looks section</td>
<td>...</td>
<td>clear syncs</td>
<td>set time signature to n/m</td>
</tr>
<tr>
<td>Set note to n</td>
<td>...</td>
<td>sumcheck all measures</td>
<td>...</td>
</tr>
<tr>
<td>Deselect all</td>
<td>...</td>
<td>...</td>
<td>deselect all notes</td>
</tr>
<tr>
<td>Set note values to whole. SHIFT=dotted.</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Set note values to half. SHIFT=dotted.</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Set note values to quarter. SHIFT=dotted.</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Set note values to eighth. SHIFT=dotted.</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Set note values to 1/16. SHIFT=dotted.</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Set note values to 1/32nd. SHIFT=dotted.</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Put notes on the nth staff down from the top. (top=1).</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Set note stems up</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Set note stems down</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

#### Look Commands

<table>
<thead>
<tr>
<th>Look Command</th>
<th>RedBug</th>
<th>YellowBug</th>
<th>BlueBug</th>
</tr>
</thead>
<tbody>
<tr>
<td>Display with accidentals and key</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Display with in-the-cracks notation</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Display in hardcopy mode (times 2)</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Display in normal mode</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Align notes and justify measures. You are now in logical mode.</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Display using note heads (half, whole, quarter, unknown)</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Display using piano roll rectangles (gives durations)</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Display notes in physical time: inches + physicalTime/n, 256 standard</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Don't show the sync sections</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Show the sync sections</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Display only voice n. Editing actions will only affect this voice.</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Display all of the voices</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

**Figure 4.2 - list of commands**
5.0 THE PROGRAMMER'S VIEW

We have seen how Mockingbird appears to the user: now we will describe how it appears to the programmer. From this point of view, Mockingbird consists of a data structure with commands to manipulate it and views to reveal it. Each of these three ideas, data structure, command, and view, will be treated in a separate section. In the data structure section, we discuss how music is represented and manipulated in Mockingbird. In the command section, we discuss how the user commands are implemented. As in the user's view section above, we separate the functionality of a command into two parts: action and interface. A command's action is its functionality independent of how it is invoked. Its interface determines how it is invoked by the user, independent of what it does. In two subsequent sections we discuss the two different views of music: graphical and audible. Each of these represents music in a form that the user can understand. We discuss in detail how these representations are produced.

Before looking inside Mockingbird, however, we give a brief description of our design philosophy. We believe that our general approach had as much to do with our success as the algorithms we used. In the following section we discuss this approach.

5.1 DESIGN PHILOSOPHY

Mockingbird is a large experimental system. This section talks about designing such systems. The emphasis is not on the static structure of designs, but on the dynamics of designing. Our approach is codified in three rules. The first two talk about how we believe a design should be broken up. Large systems need to broken up into a number of smaller systems that are more manageable. The design is implemented by elaborating each of these parts to fill them out. The design should be broken up so that one can: 1) get something running quickly 2) change the design easily, and 3) elaborate ideas to improve efficiency or functionality without having to make extensive changes. These goals lead to our first two rules:

Rule 1: separate functionality and efficiency, and

Rule 2: separate orthogonal issues.

The last rule is applicable only to problems that require "intelligent" solutions to problems. Mockingbird qualifies since we want a program that in some sense "understands" music. The rule in this case is:
Rule 3: *tools first, then heuristics.*

5.1.1 Separating Functionality and Efficiency

The first rule in designing a large system is to separate issues of functionality and efficiency. Functionality is what the system does; efficiency is how fast it does it or how much space it uses. If possible, one should design one's system so that the efficiency can be worked on independently of the functionality. This allows one to get the program running quickly and to worry later about efficiency.

For example, there may be several ways to store data, some of which are more efficient in space or time. By using procedure calls, we can separate the functionality of the data structure from the efficiency of its implementation. These procedure calls allow a client procedure to access or modify the data structure without knowing how it is implemented. The programmer can then change the underlying representations without having to change any of the client procedures.

This rule is particularly important in systems where it is not clear precisely what functionality is wanted and where, therefore, a certain amount of experimentation is required. It is pointless to waste time improving the efficiency of a module whose functionality may soon be discarded. At the same time you do not want to be trapped into a poor implementation. If these ideas are properly separated, you can postpone working on the efficiency until after the design has been proven effective.

5.1.2 Separating Orthogonal Issues By Means Of Interfaces

The second rule of design is to separate orthogonal issues. The different elements of the design should be broken into independent parts with little interaction. Each part can then be worked on without worrying about the other parts. The goal is to delay writing code until it becomes clear that it will be used. This can reduce the amount of code that is thrown away if the design has to be reworked.

Interfaces and modular programming can be very helpful in achieving this. An interface insulates a module from other modules. A client procedure can talk to a module only through the procedures defined in the interface. If the procedures are carefully chosen, their forms will not change as the design progresses (although their implementation might).

If the design is properly partitioned, one can build *facades* that do nothing. A facade is a
fully designed interface with little or no code backing it up. It will have only a small part of the functionality it should, but it can be used as scaffolding for testing another module until the facade's functionality becomes necessary. Then the functionality must be added.

An example of this in Mockingbird is the "sheet module". The sheet module is responsible for coordinating the editor with the screen. One of the most important functions of the sheet is translating between the editor's coordinate system and the screen's coordinate system. This function is dependent on how a sheet is displayed on the screen. At one point in the design, we didn't know how a sheet was going to be represented; so we hid the translator behind a procedure. The procedure is called with an object's time, pitch, and staff and it returns a screen position. This separation had a number of nice features. First of all, we could test procedures that drew things on the screen, even though the sheet was not yet implemented. All we did was to have the sheet procedure return a constant. As a result all of the objects appeared at the same place, but that was unimportant at that point in the design. Once we got them working, they did not have to be touched again. We would later change the sheet procedure, of course, but that would not affect the clients.

Later, we implemented a very simplistic sheet that allowed us to test more complicated parts of the program. This sheet had only one key and one set of staves, but was sufficient for our needs at the time. It allowed us to display simple scores on the screen. Finally, when everything else was done, we implemented a more complete sheet with key changes. We still have to add capability for dealing with variable sets of staves. The point is that the changes are independent of all the other modules, and do not cause anything else to break.

5.1.3 Tools First, Then Heuristics

At the very beginning of this project we had to make a decision about how to help the composer translate ideas into scores. There were two options open to us; build an automatic recognizer or build an amanuensis. The automatic recognizer would attempt to produce a score solely from the music as played by the user. It might allow the user to make suggestions, but basically all decisions were up to it. The amanuensis, on the other hand, would be a passive partner in the composition process. No action would take place unless the user called for it.

Clearly the automatic recognizer would be the better composer's aid, if it could be built and worked well. Looking at the problem carefully, we decided that the general automatic
recognizer was too hard for us to build. Although it would be possible to build a limited automatic recognizer, one that could handle only certain styles of music or performance, our conclusion was that no program would be able to handle all of the different types of music and all of the different styles of playing. We therefore decided on an amanuensis that would place fewer constraints on the composer's style. In exchange for this flexibility, the composer gives up speed. It takes a lot more work to produce a score with an amanuensis than with an automatic recognizer.

Of course, we can get back some of this speed by giving the user heuristics which try to do some kinds of automatic recognition. These heuristics are limited, but they can be helpful. If not, no harm is done since the same effects can be achieved "manually".

This is an example of an interesting approach to some artificial intelligence problems: first provide the user with tools to accomplish some task, and then build the heuristics that attempt to do it for him. There are two paths to a program that does something intelligent: heuristics with tools to correct mistakes, or tools with heuristics to speed up the process. The advantage of the latter approach is that early on the program will be able to do useful or interesting work with the help of the user. As heuristics are added, more and more of the intelligence is added to the program, until the user needs to do very little. With the first approach, the program will not be able to handle interesting cases until a lot of work has been put in, if ever. Besides, unless one believes one can always do the correct thing algorithmically, most of the tools that the user needs to perform the task will be required to fix the mistakes made by the imperfect heuristics. These tools should therefore be built first in any case.

5.2 DATA STRUCTURE

This section discusses how music is represented in Mockingbird. It discusses a number of different abstractions in turn and describes how they are implemented and manipulated at a low level. Some of the abstractions are data structures, others are procedures. The purpose of both is to group concepts together into modules that are easy to understand and remember. This data structure section is broken into four parts. The first discusses the most primitive level of Mockingbird, the storage system. This level is designed to hide a number of system details from the procedures that are built on top of it.

The next part talks about the piece, which is our name for music in its logical form. (See
figure 5.1.) It ignores all of the abstractions that have to do with how it appears on the screen. The piece is broken into a number of component abstractions that implement the various objects in music (chords, notes, beamed sets, tuplets and events). Each of these abstractions is examined in detail. The last part discusses the sheet, which has all of the abstractions left out of the piece. The actual data structure used here is not as important as what it does, so this part is broken down by the sheet's function. The sheet is responsible for translating between different coordinate systems, determining staffing, and determining accidentals. Each of these is discussed in turn.

5.2.1 Storage

The sole purpose of this level of abstraction is to insulate client procedures from the means of storage allocation. It consists of procedures responsible for allocating and freeing the various music objects. The procedures perform two main functions: they coordinate Mockingbird with the storage allocator, and they enforce constraints. We shall see how the latter is done after we discuss the storage configuration.

Each type of object has its own storage heap. It also has its own procedures for storage allocation and deallocation. These procedures go to the appropriate heap to allocate and deallocate objects. The heaps are implemented by system primitives. Having separate heaps allows us to write procedures that iterate over all of the instances of a particular type. For example, we search through all of the chords to find those that contain a particular note.

Early on we made the decision to represent all of the objects in Mockingbird as fixed length arrays, rather than as lists. We had two reasons for this; the lack of support for lists in Mesa, and the lack of support for variable length arrays in the storage allocator. We could have implemented a form of these ourselves, but that would have required a lot of work with only a little benefit. Instead, we chose fixed length arrays. The disadvantages of this method are that storage gets wasted and that the number of items in an object is limited. For instance, every chord has space for ten or fewer notes. If the user wants more than ten notes, he is out of luck. If a chord is to have less than ten notes, then the rest of the chord's storage is wasted. A larger limit would waste more space, a smaller limit would constrain the user.

Each class of fixed length array (e.g. chord, beam, event sequence) can be manipulated by two procedures: AddNote and DeleteNote. AddNote first checks the array to see if its note is
already there. If it is, then nothing happens. Otherwise, the array is scanned until an empty slot is found. If one is found, then a pointer to the note is stored there. If not, an error message is generated. The array is then sorted, if the class calls for it (only beamed sets and sequences of events are kept sorted). DeleteNote scans through the array to find its note. If no note is found, nothing happens. Otherwise, the note is removed from the array and the array gets packed. The arrays are always kept packed, with no gaps between notes.

As mentioned above, this level of abstraction is also responsible for maintaining consistency in the data structures. Its sole contribution to this cause occurs when an object is freed. At that time, the procedures check the other heaps for structures that refer to its object. Any reference found is then removed.

5.2.2 The Piece

In this section we discuss the components of a piece: the sequence of events, the events themselves, the notes, the chords and the beams. For each of these, two things are covered: how the component appears to a client procedure, and how it is actually represented.

5.2.2.1. notes

The most fundamental component in Mockingbird is the note. All of the other components build on it either directly or indirectly. The note consists of a number or attributes and relationships. There are three types of attributes: physical, logical, and graphical. Some attributes are shared among several of the domains, others appear in just one. The physical attributes have an effect in the physical domain; they are pitch, physical time, and physical duration. The logical attributes have an effect in the logical domain; they are pitch, voice, type, and logical duration. (There is no explicit logical time, since it is determined by looking at the interactions between voices, events, and durations.) The graphical attributes affect only how the music appears on the screen. So far there are only two graphical attributes: the staff number and stem direction.

pitch- common to all three musical domains. It indicates the key that should be struck on a piano when the note is played.

physical time- determines when the note is played. It is measured in seconds from the beginning of the piece. This time may have come from the synthesizer as the music was played in, or it may be derived from the logical score (given the tempo).
physical duration- how long the note is held. This is also measured in seconds, and it is also taken from the synthesizer input.

voice- gives the identity of the voice to which the note belongs. A note belongs to exactly one voice. (All unassigned notes belong to voice 0.)

type- distinguishes which class the note belongs to; that is, whether it is a grace note, rest, or trill. (Mockingbird treats rests as notes with certain restrictions).

logical duration- how long the note is held logically. The logical duration (also called note value) is measured in beats. The possible durations are whole, half, quarter, eighth, sixteenth and thirty-second (optionally dotted).

staff number- used by the view to determine which of the staffs the note will be displayed on.

stem direction- indicates whether the stem of the note is to go up or down.

In addition to its attributes, a note can have relationships with components of each of the other classes. A note may be part of a beam, chord, or event. The only constraint on relationships is that a note can belong to at most one instance of each class of objects. For example, no note may belong to two beams. If you want a note to appear shared, a "duplicate" note must be inserted into the data structure at the appropriate point.

Some relationships may be accessed directly. Conceptually, you could find out which component a note belongs to by searching the class of those components. However, for efficiency we keep "back pointers" in the notes for each of the classes of objects except chords. Since chords are accessed infrequently, we decided to save storage by not having a back pointer. There we actually search the chord heap to find the right chord.

5.2.2.2 chords

Chords group notes "vertically". Notes that are chorded together appear in the score with a common stem. (The direction of the stem is a parameter of the chord.) This grouping affects only the display. It is not required for the editor to play music.

Externally, chords appear as unordered sets. A client procedure manipulates a chord by using one of two procedures; AddNote or DeleteNote. Chords have two constraints: every note must belong to the same event line, and a note may appear only once in all of the chords.
These constraints are enforced by appropriate action by the procedures that manipulate chords.

Internally, chords are fixed-length arrays. Notes are added to and deleted from the arrays by the procedures mentioned above. If too many notes are added to the chord, then an error message is generated. (The extra notes could just be ignored instead.) There are also special procedures for finding a chord given a note, and sorting the notes in a chord by their pitches.

5.2.2.3 beamed sets

Beamed sets collect notes (and chords) horizontally. There is no constraint on the durations or types of notes in a beam. A beam may contain rests, grace notes, and notes all of different durations. This grouping affects only the display. It is not required for the editor to play music.

Externally, beamed sets appear as ordered sets. A client procedure manipulates a beamed set by using one of four procedures; AddNote, DeleteNote, AddChord, or DeleteChord. The components of a beam are always kept sorted by time.

5.2.2.4 tuplets

Tuplets are a special class of beamed set. In addition to the other information, a tuplet also has two parameters for scaling the note values. The first indicates the number of beats currently in the beamed set, and the second gives the number of beats that the beamed set replaces. For example, a triplet is a tuplet in which three notes replace two of the same value. When the triplet is played, those notes are played fifty per cent faster so they all fit in. The amount of scaling is a function of the two numbers. The actual numbers used are not important, only their ratio.

Tuplets can be drawn on the screen in two different ways. There is the normal way with beams and a special way using phrase marks. This happens, for example, when rests form part of a tuplet. Rests do not have stems (and therefore cannot have beams), and so the publisher uses a phrase mark. We do the same thing. In either case, tuplets also have a number above them that indicates how much scaling is applied. This number is usually but not always the count of notes in the tuplet. It actually counts beats rather than notes. An example that shows this is the case of a triplet made up of a half note and a quarter note. This case has two notes but three beats, and so is marked as a triplet.

5.2.2.5 events

Events mark a place in the piece where "something happens". There are two types of
events; measure lines and sync lines. Measure lines are used as markers in the score. The sync
lines indicate points at which a number of notes are synchronized (hence the name).

Events have attributes, the most important of which is time. The time of an event indicates
where the event is to be displayed. We thought it important to put this information on the
event rather than the note. A note finds out its display "time" by checking its event. There are
two benefits to this: notes that have the same time can be moved as a group, and the times of
notes are always consistent (that is, if two notes are in the same event, they always have the
same time).

There are two other event attributes; time signature and key signature. These are discussed
in sections 5.3.2.5 and 5.3.2.6.

5.2.2.6 voices

The voices of a piece are actually phantom objects; they do not have representations as
objects in the data structure. Instead, each note is labeled with the name of its voice. Most of
the procedures know about voices, but they use the labels on the notes for their work. Voices
do not have any meaning beyond these labels.

5.2.2.7 the sequence of events

The sequence of events is the backbone of the piece. It is a collection of all of the events
in the piece, ordered by their times. If a procedure works on the piece, it does so by iterating
over this sequence.

Externally, the sequence of events appears, as a sequence. The sequence can be
manipulated by a number of procedures; AddEvent, DeleteEvent, CopyPiece, and ReplacePiece.
CopyPiece makes a copy of a section of the sequence. This copy includes all the beams and
chords associated with the section. ReplacePiece replaces a section of the sequence with another
sequence. It is the basis of all sequence editing actions. If the section is empty, then this is
equivalent to an insert. If the sequence is empty, then this is equivalent to a delete.
ReplacePiece can take its sequence from CopyPiece or from the synthesizer keyboard. The latter
action is called recording.

5.2.3 The Sheet

The sheet is the template upon which the piece of music is represented graphically. To the
user it appears as staves on a sheet of paper. The sheet therefore interfaces the editor with the
screen. It has three functions as a data structure: translating between different coordinate systems, determining staffing for notes, and determining accidentals for notes.

5.2.3.1 translating between coordinate systems

The screen and editor use different coordinate systems; the sheet translates between these. The screen uses an (x,y) coordinate system, where x and y measure the horizontal and vertical positions of an object. The editor uses two coordinate systems: (time, offset) and (time, staff, pitch). (The offset gives the vertical position of an object in inches from the top of the staves; the staff, pitch pair gives the vertical position relative to a particular staff.) Whenever the object is drawn, its musical coordinates must be translated into a position on the screen. Whenever the user selects a position on the screen, the coordinates of the mouse must be translated into music coordinates to find an object. The sheet handles both types of translations.

The first type of translation, music to screen, is unambiguous. Internally, the editor "thinks of" the music space as two dimensional, with time on one axis and offset (or staff and pitch) on the other axis. A score is a sequence of events on top of a long set of staves. The staff set contains all the information about how the staves are to be displayed; their relative position, the height of the set, and the offset to the next set. We call a particular staff set a line. To get a score on the screen, it must first be broken into a number of lines. An object's position is given by the line, and position within the line. The sheet does this by having an array of lines. Each line has a position, starting time, and staff set. The line's position is its place on the conceptual screen's infinite plane. It is a function of the heights and offsets of all of the lines that have gone before. The line's starting time measures the line's position in music coordinates. It tells where the line begins in the score. The start time is a function of the lengths of all of the lines that have gone before.

Once the sheet has decided which line an object is on, the object's position on that line must be determined. The sheet does this by referring to the associated staff set. Given the staff and the pitch, the sheet can determine the object's relative position on the staves. The object's absolute position is the sum of this relative position and the line's absolute position.

The second type of translation, screen to music, is ambiguous. A point on the screen can map into several points in the music. Consider a point halfway between two lines on the screen. Which line does it belong to? The answer to this will determine the point's time.
Unfortunately, the point could belong to either line. Similarly, a point between two staffs may belong to either staff.

The purpose of this translation is to find that object in the data structure which the cursor is designating, so we can usually get around the ambiguity by trying all of the possible objects and choosing the one that is closest to the cursor. This handles the intent of this sort of translation.

5.2.3.2 determining staffing

The sheet does not always display an object on the staff specified by the user. This is because the set of staves may have changed since the object was given a staff. Consequently, the sheet uses a special scheme to determine which staff in the set of staves the user intended.

Here is a simple example that demonstrates problem. Suppose the user decided to put a note on the second staff of a three staff system made up of two treble staffs and a bass staff. The note now knows that it’s staff is staff "2". Later the user changes the staff set to be a treble and two basses. Which staff should the note be displayed on now? Staff "2" is now a bass staff, but the user put the note on a treble staff. Suppose that the editor changed the note’s staff to staff "1" so that it would appear on the treble staff. What happens if the user goes back to the original staffing system? Which staff does it go on then, "1" or "2"? Both are treble clefs, so there is no clue there. How do we handle these problems?

We decided that notes should always be displayed on the same type of staff, no matter how the sheet is changed. We also decided that notes should preserve their staff in a particular set of staves, so that when the user returns to a set of staves, it appears as it did before (unless he made explicit changes). Both goals are accomplished by the following scheme.

Let each set of staves have room for six staffs. The most general set of staves has six staffs: a super-treble (two octaves above normal treble), two trebles, two basses and a super-bass (two octaves below normal bass). These are labelled "1" through "6" respectively. Smaller sets of staves are a subset of this set. For example, the staves with two trebles and a bass has staffs "2", "3", and "4". The staves with a treble and two basses has staffs "3", "4", and "5". Whenever you wish to draw an object on a missing staff, put it on the nearest staff. If there are two staffs equally near (for instance, when there are staffs "1" and "3", but no staff "2"), put it on the staff towards the center (towards "3" or "4").

Let us see how this works in the example given above. In the example, the user puts a note
down on the staffing system with two trebles and a bass. When he specifies that the note is on the second staff, the note is actually told that it is on staff "3". This locates it on the second staff of that staff system. When the user changes the set of staves to a treble and two basses, the note is still displayed on staff "3", which is now the first staff of that staff system. If the user switches back to the original staves, he finds that everything was as it was before.

5.2.3.3 determining accidentals

The last function of the sheet is to determine the accidentals that appear to the left of notes. The accidental chosen depends on two things; the key of the section and the notes that precede the note in question within the measure. That the key is necessary is fairly obvious; the whole purpose of a key is to change the set of notes that do not need an accidental. Preceding notes affect the need for accidentals because of a convention of music notation: an accidental affects all of the notes at that pitch until the next accidental or measure line. Hence, accidentals are not displayed if there is a previous note in the measure at that pitch: the former accidental will eliminate the need for this one. However, accidentals must be reasserted if there are conflicting accidentals preceding them.

This function can be broken into two parts: finding the accidental given the key, and seeing if the accidental is affected by prior notes in the measure. The first part is accomplished with a two dimensional array of keys and pitches. This array has three possible entries: sharp, flat, or null. The null entry indicates that no accidental is necessary. The accidentals are determined by finding the pitch under the appropriate key. The second part begins once we know this accidental. The score is scanned backward from the note in question until a measure line or a note at the same height is found. If a measure line is found first, the accidental is not modified. Otherwise the accidental is changed according to the accidental of the note on the same line. If the accidentals are the same, then there is no need to assert the accidental again and no accidental is drawn. If the accidentals are different, then an accidental will be drawn, even if no accidental was necessary in the first place.
5.3 THE GRAPHICAL VIEW

The graphical view is the view of the data structure that appears on the Dorado screen. There is a view program in Mockingbird that builds something like a score from the data structure. To accomplish this, the program needs to know how to draw the different components of a score and how to arrange them into something a musician will understand. The quality of the score is not a critical issue. We are not trying to produce something publishable, only readable.

The following sections will describe how we draw the different components and put them together. First, however, we will examine the tools at our disposal.

5.3.1 Graphics Package

The foundation upon which all of Mockingbird's graphics procedures are built is the Mesa graphics package. This package provides a device independent interface through which one can handle graphics. Only one set of procedures is required for printing on the printer and drawing on the Dorado screen. The graphics package consists of a number of graphic actions and a context. A particularly important feature of the package is that a multiplicity of fonts is easily accommodated. This enables us to use a special font that contains characters for the common musical objects.

5.3.1.1 basic commands

The basic graphics actions in CedarGraphics are; DrawLine, DrawRectangularArea, DrawPolygonalArea, DrawSpline, and DrawText. Both DrawRectangularArea and DrawPolygonal Area result in a filled in area, not an outline.

DrawLine- takes two points and draws a line between them.

DrawRectangularArea- takes two points and uses them as the lower left and upper right points of a rectangle.

DrawPolygonalArea- takes an arbitrary number of ordered points as the vertices of a polygon.

DrawSpline- takes a number of points and draws a Beysian spline curve through them.

DrawText- takes a character from an arbitrary font and paints it at the place specified.

5.3.1.2 concept of context

Graphic devices are accessed in graphics package through a context. The context is a view
onto the device. It translates the graphical actions mentioned above into actions appropriate to the device. The context is also responsible for keeping track of the attributes of the view. (The view can be scaled and rotated; the texture and color of the paint brush can be changed.) Several contexts may be attached to the same device.

5.3.1.3 the music font

In addition to the above commands, Mockingbird uses a special font of music characters. These characters provide a number of fixed objects that can be used as building blocks. Variable objects can be constructed from these fixed objects and the above commands. For instance, a note is constructed from a note head character and a flag character. The two characters are connected with a variable length line. There are also special characters for all of the accidentals, as well as the bass and treble clefs.

5.3.2 Displaying Music Objects

The standard score has a number of objects that are unique to music. Mockingbird must be able to draw each of these. As much as possible, we have tried to stick to the rules of standard music notation. The following section will discuss how sheets, notes, chords, beams, keys and time signatures are displayed.

5.3.2.1 sheet

The sheet is the grid upon which all of the objects are placed. It appears on the screen as lines of staves. The staves are a set of staffs. To draw a staff, five equidistant horizontal lines are layed down. To the left, the appropriate clef is displayed (treble or bass). Following the clef is the key signature, if any. (Key signatures are repeated at the beginning of every line.) All of the staffs of a stave are connected together by a vertical line at the extreme left. (The line that normally appears at the right is a measure line; it will be drawn when the rest of the score is drawn). This procedure is repeated for each of the lines on the screen.

5.3.2.2 notes

The most common graphical object is the note. It consists of a note head, stem and flag. To display a note, we need to know its time, value, pitch and staff. From these we can usually determine where the note head is to be drawn. The note value is needed for the note head and flag, if any. Any note value smaller than a half uses a filled in oval for its head. The half note's head is an empty oval; the whole note's head is an empty oval with thick outlines.
(People are often surprised to discover that the whole note and half note have different shapes. Careful inspection reveals that they are indeed quite different.) The stem of a note is always at least three and a half ledger lines long. It always reaches at least to the middle of the staff. The flag used is determined by the note’s value. An eighth note has one flag, a sixteenth note two, a thirty-second note three, and a sixty-fourth note four. If the note lies above or below the staff, then ledger lines will have to be drawn up or down to the note’s head. The procedure asks the sheet for the appropriate accidental and, if there is one, draws it a predetermined distance in front of the note’s head. If the note is dotted, a dot follows the head. If this dot would fall on the line, it is moved up to the space above the line for visibility.

5.3.2.3 chords

Chords group notes vertically. All of the notes in a chord share one stem (and flag). The chord is responsible for drawing the stem and the flag. The constraints here are the same as for an individual note: the stem is to be at least three and a half ledger lines long, the stem always reaches to the middle of the staff if it is going that way. After it has drawn the flag and stem, the chord tells the notes to draw themselves.

5.3.2.4 beams

Beamed sets connect notes and chords horizontally. A beamed set knows the origin and tilt of its beams. The first thing it does is to determine the number and placement of its component beams, depending on the note values it holds. (If the beams wraps around the end of a line, two sets of beams will have to be drawn.) Once it has done that, it computes the stem heights for each of the notes and tells the notes or chords to draw themselves appropriately.

Tuplets are a subclass of beamed sets. In addition to specifying the beaming, the tuplet must draw a number centered under or over the beam, depending on whether the stems run up or down. There is a special type of tuplet which is drawn something like a phrase mark. This is fairly easy to draw, since it does not require multiple beams.

5.3.2.5 key signatures

Key signatures appear in the score as a sequence of sharps or flats. This sequence represents the key; it determines the pattern of accidentals for the notes between it and the next signature. There are fifteen key signatures: seven with sharps: seven with flats, and one with no accidentals at all. The key signatures vary according to the clef they are displayed on.
The sharp and flat key signatures fit into patterns that make displaying them easy. All key signatures with a particular accidental are a subset of the key signature with seven of those accidentals. The difference is that the trailing accidentals are suppressed. Thus, the key of E, which has four sharps, is made up of the first four accidentals of the key of C#, which has seven sharps. To draw all of the key signatures, one has to know only how to draw the key signatures with seven accidentals.

5.3.2.6 *time signatures*

Time signatures appear in the score as two numbers, one above the other. The numbers themselves come from the special music font. The top number can be anything in the range of 1 to 16, the bottom number must be a power of two. Thus, 2/4, 5/4, 7/8 and 12/16 are all legal time signatures, but 2/3 is not.

5.3.3 Layout

An important aspect of displaying music is deciding where objects are to be placed. Music notation gives the copyist some freedom in deciding the spacings between certain objects on a sheet of music. He can specify the distance between successive lines, successive staffs, or successive events. In Mockingbird, the first two are determined by constants built into the program and the third is determined by a heuristic. The current section is devoted to exploring this heuristic.

There are two domains where event layout is important: the physical domain and the logical domain. In the physical domain, each event has an absolute place in the score which is determined by its time of occurrence. In the logical domain, each event's place in the score is relative to the prior event.

5.3.3.1 *physical domain*

In the physical domain there is a linear correlation between when an event is played and where it appears on the screen. To layout a score, the editor first pretends to play it using the procedures described in the next section. These procedures determine the time of occurrence for each event. The times are then multiplied by a constant to get positions in the score. (The user provides the constant.) There is no guarantee that measures will align with the edges of the screen.

This scheme has two problems: positioning events that are not played, and displaying
objects that start on one line and end on the next. So far, the only events which are not played are measure lines. We arbitrarily place them a fixed distance before the event that follows the measure line.

Since measure lines do not always fall at the edges of the screen, beams and phrase marks may start on one line and end on the next. This means that the procedures that draw these objects must detect this situation and handle it. This type of breakage can be detected by examining the notes' positions on the screen. If one note is at the right of the screen and the next note is over at the left, then the beam or phrase mark has wrapped around. If this happens, then two of the objects are drawn, one on each line. These objects are extended beyond the edge of the lines to indicate that they are logically connected.

5.3.3.2 logical domain

There are three conflicting goals that affect the spacing between events in the logical domain. 1) The spacing should be associated with the duration of the former event. (There should be more distance after a half note than after a sixteenth note, although not necessarily eight times as much.) 2) The spacing should be as dense as possible. We want to get as much music on the sheet as we can, so as not to waste paper. 3) However, the score should be readable. Objects should not overlap or be cramped.

Mockingbird has a complicated procedure that attempts to satisfy these goals. First, the logical distance between two events is determined. This distance gives us a nominal spacing for the events, assuming that there are no complications. Next, if there are any accidentals or dots between the two events, the nominal spacing is adjusted to allow enough room for these objects. If one of the events is a measure line which has a time or key signature on it, still more space is allocated.

We use logical distances rather than durations since there may be no duration between the two events. This would happen if successive events did not have any voices in common. For example, suppose that voice 1 has a quarter note A and a half note B, while voice 2 has a half note C followed by a quarter note D. There are three events: A and C together, B by itself and D by itself. Events B and D do not share any voices. It would be incorrect to use B's duration of a half to determine the spacing between B and D, since it is in the wrong voice. What we really want is the logical distance between the two events, which in this case is one quarter.
A complication to the above scheme for determining spacing is that measure lines should appear aligned with the screen. That means that there should be a measure line on the left and right sides of each line. This is "justification" for music. Our solution is first to determine the spacing for the notes as outlined above, assuming that we wanted things as dense as possible. When the spacings have been calculated, we stretch each line to make it fit. If two and a half measures fit on the first line, we stretch the first two measures linearly so they fit exactly, pushing the half measure onto the next line. This is repeated for successive lines until we run out of measures.

5.4 THE AUDIBLE VIEW

The audible view is the view of the data structure that the user gets through the synthesizer. It is concerned with how a score sounds rather than how it appears. There are two aspects of the audible view: playing and recording. Playing produces an audible representation from the data structure. It attempts to produce what a musician would produce if he were to play the piece. Recording takes in music as the user plays it on the synthesizer and produces a pianoroll. (The user is then responsible for converting the pianoroll into a score, if he desires.)

In the following sections we discuss the hardware and software that support the audible view. At the bottom of the process is a Yamaha CP-130 synthesizer. The Yamaha is treated as an input/output device by the Dorado. Special microcode handles the low level work for the Dorado. (Both the synthesizer and the microcode were obtained or developed for this project.) Above the microcode are a pair of processes for playing and recording music. These processes run independently from, and in parallel with, the rest of the editor. They have a higher priority than the editor, so that playing and recording take precedence over other user actions. This means that the user can edit a section while it is being played without disturbing the playing. Both processes work indirectly with the synthesizer through the microcode. They convert back and forth between keystrokes and the data structure. Generally the procedures work on notes in batches because the microcode has limited buffering which the music processes quickly overrun.

5.4.1 The Synthesizer and Microcode

5.4.1.1 the synthesizer

The synthesizer consists of 76 keys and 76 tone generators. (That this number is not 88 is a detail of the particular keyboard and is of no significance.) Each key has two switches, A and B.
When a key is struck, switch A opens before switch B closes. When the key is released, switch B opens before switch A closes. A key may be in any of three states: switch A closed and switch B open (key up), both switches open (in transition), and switch B closed and switch A open (key down). These two switches control the behavior of the key's tone generator. The length of time that switch B is closed determines the duration of the note. The difference in time between the opening of A and the closing of B determines the loudness of the note.

The computer reads from and writes to the synthesizer by observing and mimicking these switches. This enables it to record keystrokes from the synthesizer keyboard and to playback music through the synthesizer's tone generators.

5.4.1.2 the microcode

Special microcode handles recording and playing through this interface. It can play and record at the same time. When recording, the microcode scans the entire keyboard every 78 milliseconds looking for state changes. Any transitions and their times are put into a buffer which is passed to the higher level software for processing. When playing, the microcode processes another buffer which contains similar transition-time pairs. The key states are set when the appropriate moment arrives.

5.4.2 The Player

The player is a program whose job is to convert Mockingbird's representation of music into transition-time pairs for the microcode. There are three parameters for this: speed, voice, and domain. The speed is specified by the user as a multiplicative constant and determines the time scale, i.e. how fast the music is to be played. The voice specifies which collection of notes should be played. The domain indicates whether the music is to be played from the physical domain or the logical domain. The player plays music using whichever voice and domain the user chose for the graphical view. This means that the user always hears things as they appear on the screen.

Transforming a piece from Mockingbird's internal representation into transition-time pairs is not a straightforward task because the piece may be a mixture of physical and logical representations. Each note has two representations: physical and logical. Notes from a pianoroll do not have logical representations until the user assigns logical durations. Notes inserted from the pop-up menu do not have physical representations unless the user asks Mockingbird to
derive them. This means that the player may encounter notes that have no representation corresponding to the current domain (e.g., the player may encounter physical notes while playing in the logical domain). Something special will have to be done to handle these notes.

Mockingbird handles these abberations in one of two ways: by deducing the necessary information from the context, or when this fails, by temporarily switching domains. If any of the notes in an event has the proper representation for the current domain, then Mockingbird can deduce the correct time for the rest of the notes. This is because events are defined to be collections of objects that all occur at the same time. If the time for one note can be determined, the time for all of the notes can be determined. The missing durations are taken from the other representation. If none of the notes has the proper representation, then Mockingbird temporarily switches domains and plays the event in the other domain. Mockingbird switches back as soon as it encounters a note with the proper representation.

This means that Mockingbird needs only two procedures for playing music: one that plays physical music in the physical domain, and one that plays logical music in the logical domain. Some care must be taken when switching from one domain to the other, since there are two different metrics for time. All of this is discussed in the following sections.

5.4.2.1 physical music in the physical domain

Mockingbird scans the score looking for physical representations for the notes. If the physical representation for a note cannot be found or deduced, then it switches to the logical domain (see below). To play a physical representation, the times and durations are scaled by the speed constant and a set of transition-time pairs are formed for the microcode. The microcode needs three transitions for every note: one for when the key is struck, one for when the key hits bottom, and one for when the key is released. The first transition occurs when the note starts. The loudness of the note determines the time between the first and second transitions. (Currently, all of the notes are played at a fixed volume.) Finally, the duration of the note determines the time between the second and third transitions. These transition-time pairs are put into a buffer and given to the microcode to be played on the synthesizer.

5.4.2.2 logical music in the logical domain

The playing times of notes in the logical domain are derived from the interactions between the logical durations and voices in the piece. To find out the playing time for a particular note,
we must determine the time of its event in beats measured from the beginning of the piece. Once the logical times are known, a physical representation can be derived using a beats-to-seconds conversion factor. It is then a simple matter to generate the transition-time pairs as described above.

One might think that the time of an event is found by summing up all of the notes prior to it by voice, but this does not take into account gaps in the voices (which are commonplace) or errors such as erroneous note values. We use a slightly more complex scheme that takes these problems into account. Sums are kept for each of the voices, and events are processed in turn. When an event is processed, the procedure determines which voices participate in it and identifies the one with the largest sum. This sum is the logical time of the event. All of the other sums are then set to this sum. After that, the duration of each voice in the event is determined. If there are several different durations, the smallest is used. (The event should have a voice with several durations only if the user has not properly voiced the piece. We have found by experience that the best approximation to the user's intent is to take the smallest of the durations.) These durations are added to the sums, and the whole process is repeated for the next event.

5.4.2.3 problems caused by switching domains

We mentioned above that the player must occasionally switch from one domain to the other to handle notes that have no representation for the chosen domain. This switch is complicated by the different meanings of time in the two domains. For example, what should Mockingbird do if it is thirteen beats along in the logical domain (translated to thirteen seconds) and it encounters a physical note with a time of three seconds? Twenty-seven seconds? Some adjustment must be made to bring the two domains into alignment. Mockingbird deals with this by deciding when the note should occur relative to the immediately prior note. It finds the logical duration for that note, and puts the next note where the duration ends. This algorithm is not foolproof, and occasionally the user will hear an unnatural pause or hiccup as the domain changes. Fortunately, such minor interruptions are not serious problems and they go away as the user finishes editing the score.

Let's look at the example above again using this heuristic. Suppose that the last note in the logical domain had a half beat duration and began on the thirteenth beat. This indicates that
the next note should begin at thirteen and a half beats, which translates to thirteen and a half seconds. If the physical note had a time of three seconds, then ten and a half seconds are added to its time and to the times of all the subsequent physical notes.

As another example, suppose that Mockingbird is three seconds along in the physical domain when it encounters a logical note. Suppose also that the last note in the physical section began at three seconds and was physically held for a half second. When should the next note occur? One might say at three and a half seconds, but we have found that a physical duration is usually smaller than the corresponding logical duration (pianists rarely hold down a note for the full time allotted) and so we would put the next note around three and three-quarter seconds.

5.4.3 The Recorder

The recorder's job is to convert the microcode's transition-time pairs into a pianoroll. The transition-time pairs are all mixed together along with spurious transitions caused by switch bounce. (Note A might be struck, followed by note B and note C. Then note B is released, and then note C bounces and is released, and then finally note A is released.) The transitions come already sorted by time. The recorder must group them into notes, filtering out the spurious transitions.

To process the transitions, the recorder updates a state table which has all of the keys. The recorder examines a transition, decides what key it applies to, and changes the state of that key appropriately. If the transition is spurious, it is ignored. When a note is completed, it is appended to a private data structure and the key's state is cleared. When the user is through recording, the data structure is inserted into the score at the place indicated by the user.
Figure 5.1- Model of the Data Structure
6.0 CONCLUSIONS

Mockingbird stands now as a useful system. Several people have started to use it, and are quite pleased with its performance. There is a small library of about a hundred pages of music stored in the PARC file system that contains such things as Beethoven's Moonlight and Appasionata Sonatas and Bach's Little Fugue in G Minor. The people that have used it generally find that it takes somewhere between fifteen and fifty minutes to score one sheet of music, depending on the complexity of the music and their skill as pianists.

The actual program consists of ~7000 lines of Mesa code in seven modules. The program is not too complex, just detailed. Mockingbird uses 30K of memory for the program and about 8K of memory for each sheet of music. It responds quickly to the user actions; the slowest action is to redraw the score completely from scratch (1-2 seconds). Minor modifications to the score appear instantaneous.

The remainder of this section is devoted to discussing the work that needs to be done to finish Mockingbird as a composer's aid, and implications for Mockingbird as a composer's aid and a teacher's aid.

6.1 LEFT TO DO

While it is currently a usable system, there are a number of features that clearly need to be added to Mockingbird. We group the features into two categories: musical features and program features. Musical features are items that need to be added to complete Mockingbird's ability to represent music. Program features are items that need to be added to increase the editing power of Mockingbird.

6.1.1 Musical Items

Currently, Mockingbird can handle only a subset of standard music notation. Although this subset is sufficient to enable the composer to capture his ideas, we want to add a number of features to expand Mockingbird's range. These features can be broken into three groups: features that facilitate performance (embellishments and performance cues), features that facilitate understanding (slurs, ties, and spelling), and features that facilitate reading (changing staffing and graphical adjustments).
6.1.1.1 embellishments

Mockingbird presently has no ability to handle the standard embellishments of music: grace notes, trills, rolled chords, etc. Adding these to Mockingbird means that Mockingbird would have to be able to represent, display, and play each one.

6.1.1.2 performance cues

Mockingbird has no ability to handle any type of performance cues. There are three types of performance cues: those that affect how loudly notes are played, those that affect their duration, and those that affect the tempo of a piece. Mockingbird should be able to represent, display, and ideally even utilize (in playing) the standard notations for these items.

Loudness cues vary according to whether they apply to a section or a note, and whether they indicate a constant loudness or a varying one. Sectional cues are cues such as \textit{f}, \textit{pp}, or diminuendo. The first two indicate sections of constant loudness, the third indicates a section of decreasing loudness. Note cues such as accents are cues that apply to a particular note or chord.

There are two cues that affect a note’s duration: staccato and legato. Staccato means that the note’s duration is artificially shortened. It is indicated by a dot above the note. Legato means that the note’s duration is artificially lengthened to its full logical value so that it connects to the next note (or chord) of the voice. It is indicated by a dash above the note.

Tempo cues vary according to whether they apply to a section or a note, and whether they indicate a constant tempo or a varying one. Sectional cues include markings such as \textit{andante}, \textit{allegro}, or \textit{accelerando}. The first two indicate sections of constant tempo, the second indicates a section of increasing tempo. Note cues are cues such as a fermatta that apply to a particular note or chord.

6.1.1.3 ties and phrase marks

Ties and phrase marks appear as slightly curved lines on the score. Ties indicate that two successive notes with the same pitch should be thought of as one note with a duration equal to the sum of the two. The effect of this is that the second note is not struck. Phrase marks group notes and chords into phrases.

6.1.1.4 spelling

Spelling is deciding how a particular pitch will appear on the score. For example, the note between C and D may appear as a C sharp or a D flat; a G may sometimes be shown as an F
double sharp. Currently, Mockingbird chooses one of these with a very simple algorithm which does not always produce the musically "correct" spelling. Eventually, we want the composer to be able to change a note's spelling by hand. A harder problem would be to make a heuristic speller that guessed at the spelling for the composer, based on the rules of harmony (the problem being determination of the harmonic context).

6.1.1.5 sectional staffing

Currently, the view's sheet applies to the entire score; there is no way to have two staffs in one section and three staffs in another. This will be changed so that the user can change clefs on a staff anywhere he pleases, and also indicate that certain staffs are octava (shifted up or down one octave).

6.1.1.6 graphical adjustments

Mockingbird's display algorithm has a number of problems that need to be fixed. These problems are outright violations of the rules of standard music notation. One is that there is no way to handle notes in chords with adjacent pitches (See figure 2.1.5). In the present algorithm, these notes overlap. In standard music notation, the notes are separated by the stem. Another is that adjacent chords overlap. Standard music notation separates these slightly to make things clear (See figure 2.1.6). There are more adjustments such as these (e.g., making sure adjacent accidentals do not overlap) that we expect to add to make the display algorithm better.

6.1.2 Program Items

There are several items that we would like to add to Mockingbird to make it a better editor. In particular it needs multiple windows, an ability to copy structures, and an ability to undo commands.

6.1.2.1 multiple windows

Eventually Mockingbird should have multiple windows and subwindows, viewing the same or different files. These would allow the user to copy music from one score to another, or from one section in a score to another section that is far away. The user would need to be able to create, destroy, and move the boundaries of each type of window.

6.1.2.2 copying structures

Mockingbird needs a reasonable feature for imposing the structure of one part of a piece on the notes of another part. A great deal of music contains structures (such as the patterns of
durations or beaming) that are repeated over and over again in the piece. The only difference is in the pitches of the notes. The editor would be much more useful if the user could specify these structures once and then apply them over and over again.

6.1.2.3 undo

Many times in using Mockingbird we have wished that we had the ability to undo commands. However, this is a hard feature to build into an editor since each possible action has to have a converse action or keep a history of prior state. Whenever an undo is executed, the editor must reverse the flow of time, undoing all of the actions and reverting to the prior state.

6.2 ADVANTAGES OF MOCKINGBIRD

There are four main advantages to Mockingbird over more traditional means of writing music: faster capture, easier editing, better error detection, and higher quality printout. We discuss each of these below.

6.2.1 Faster Capture

The composer can enter music into Mockingbird as fast as he can play it. The music entered in this way is not in its final form, but it has been captured, and it can be replayed and edited. The composer does not have to worry about losing the idea while he is working on other ideas. Mockingbird's performance here can be contrasted with the use of pencil and paper, where it takes many times longer to capture a complex idea even if the composer uses shorthand and writes as fast as he can. The composer can capture music quickly with a tape recorder, but then the music is difficult to access and hard to edit.

6.2.2 Easy Editing

Editing is easier with Mockingbird than with traditional means for much the same reasons computer text editors are useful. The composer can change and rearrange a piece by issuing a few commands. Editing on paper requires erasing sections and redrawing them. This is slow and often produces a messy, confused manuscript even when a change is small. Editing a tape requires the composer to cut out sections physically and tape them together. If the user wants to make a small change, he has to re-record an entire section.

6.2.3 Better Error Detection

Mockingbird allows the composer to check his work by playing it back. Since Mockingbird plays the music as it is written rather than as it was intended, the composer will be able to catch
transcription mistakes as he listens. This is because it is much easier for a musician to hear errors than to see them. Mockingbird also has a constraint checker that checks the number of beats in a measure. These two features help the composer obtain a correct score quickly.

In passing, note that this ability of Mockingbird addresses what is a current problem in the publishing world: many pieces get published with blatant errors in them because there is no sure and easy way to detect the errors. Publishers generally hire accomplished musicians to check a piece by playing it, but musicians often play what they expect to see rather than what is there, and so errors are overlooked.

6.3 A TEACHER’S AID - ANOTHER APPLICATION

As we worked with Mockingbird, we discovered that it had a potential use as an aid in teaching music. We found that Mockingbird could help us read, understand, and perform music better. We find it an interesting speculation that music teachers might be able to use Mockingbird as a tool to help their students develop these skills.

6.3.1 Reading

When a section of music is played, a cursor points at the notes currently heard. The student can use this to associate what he sees on the screen with what he hears. He can also learn how different ways of showing music are similar by manipulating the view of the score.

6.3.2 Understanding Music and Music Notation

Mockingbird allows a user to view music in several different ways. In particular, it allows one to listen to the various voices separately, and it may be that this allows the student to obtain a better grasp of the music. It also seems likely that the ability to play things at widely variable speeds can enhance his understanding of the overall structure.

6.3.3 Performing

It seems possible that Mockingbird could help a student improve his performance by keeping an exact record of how he played in the form the pianoroll. This can allow a teacher to point out mistakes that the student made, and show him in detail what he did wrong. The student can compare the pianoroll of his performance with that of the teacher’s or the computer’s performance, thus gaining a better understanding of his mistakes.
6.4 IMPLICATIONS

Mockingbird's success has demonstrated that solving problems with an amanuensis is a useful and powerful concept. We speculate that this approach might be helpful with other problems that have yet to be solved by straightforward algorithmic means. We hope that our success in the domain of music will encourage people to try this approach in other domains.
BIBLIOGRAPHY

[1] William Buxton... The Evolution of the SSSP Score Editing Tools
   Computer Systems Research Group University of Toronto

   Composition Technical Report CSRG-97

   Computer Music Journal Vol. III, No. 1

   Journal of the Audio Engineering Society Vol 20 No. 1

   Computer Music Journal April 1977

   Microporccessor-based CRT Music Terminals (trade literature)

   Technology Review Nov. 1979