Machine Listening and Composing:
Making Sense of Music with Cooperating Real-Time Agents

by

Robert James Rowe

M.A., Music Composition, University of Iowa, 1978
B.M., Music History & Theory, University of Wisconsin, 1976

Submitted to the Media Arts and Sciences Section, School of Architecture and Planning, in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
at the
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 1991

Signature of Author .....
Media Arts and Sciences Section
May 3, 1991

Certified by . . . .
Tod Machover, Associate Professor of Media and Music
Thesis Supervisor

Accepted by . . . .
Stephen A. Benton
Chairman, Departmental Committee on Graduate Students

Machine Listening and Composing: Making Sense of Music with Cooperating Real-Time Agents

by
Robert James Rowe

Submitted to the Media Arts and Sciences Section, School of Architecture and Planning, on May 3, 1991, in partial fulfillment of the requirements for the degree of Doctor of Philosophy at the Massachusetts Institute of Technology

Abstract

Cypher is a real-time interactive music system with two major components: a listener and a player. The listener analyzes streams of MIDI data. The player uses various algorithmic techniques to produce new musical output. Both components are made up of many small, interconnected agents, operating on several hierarchical levels. The listener classifies features in the input and their behavior over time, sending messages which communicate this analysis to the player. A user of *Cypher* can configure the player component to react to such messages, where a reaction is the execution of compositional methods producing new music in response. Features characterized include speed, density, dynamic, harmony, and rhythm.

A graphic interface allows the specification of relations between feature classifications and types of response. Collections of relations can be saved and recalled during performance by a score orientation section which tracks human performance and executes state changes at predetermined points in the score. Further, an internal critic analyzes and alters *Cypher*'s own output, representing a set of programmed aesthetic preferences, and ensuring a consistency of style in the program's responses.

Cypher plays composed music in a manner that is sensitive to live human performance cues. It is able to analyze and respond creatively to unknown music. Finally, it can compose without input, using algorithms to either transform remembered material, or generate new musical output.

Thesis Supervisor: Tod Machover
Title: Associate Professor of Media Arts and Sciences

Acknowledgments

Cypher was written at the MIT Media Laboratory, and would certainly not have taken the same form anywhere else. The combination of people, facilities, and atmosphere provided a constant stimulus and challenge: above all, it has been a great place to make and think about music. Particular thanks go to my thesis supervisor, Tod Machover, for his unwavering support and encouragement of my research and composition over the last four years. A good teacher, critic, inspiration, and, above all, a good friend. I want to thank the other members of my committee as well: Marvin Minsky for his challenges and insight about how to get a computer to make good music; and John Harbison for sharing his thoughtful and penetrating perspective on the relation of my work to music composition and theory.

Barry Vercoe helped, supported, encouraged, and improved my work in many ways I greatly appreciate. Thanks also to my colleagues at the Lab – I was blessed with a great musician and provocative debater as an officemate in Michael Hawley. My other (upstairs) officemate, Joseph Chung, was always a helpful source of code, ideas, discussion, and experience. Other members of the Music & Cognition Group read drafts, debugged talks, and made me think: Alan Ruttenberg, Mike Travers, Mary Ann Norris, David Rosenthal, Shahrokh Yadegari, Tomoyuki Sugiyama, Adriano Abbado, Dan Ellis, Lee Boynton, Jim Davis, Andy Hong, and Greg Tucker. I was fortunate enough to profit from the thought given to my work by many performers, researchers, composers, and friends: Richard Teitelbaum, George Lewis, Steve Coleman, Muhal Richard Abrams, Cort Lippe, Bruce Pennycook, Wim Boogman, Christopher Oldfather, Michael Century, Garry Beirne, Paul Berg, Roger Dannenberg, David Wessel, Miller Puckette, Barbara Pritchard, and John McDonald.

Thanks to my wife Tamara for letting me use the computer once in a while, and for generally making my life a living heaven. This thesis is dedicated with love to my parents, for showing me how to live.

This report describes research done at the Media Laboratory of the Massachusetts Institute of Technology. Support was granted by the Yamaha Corporation.

Contents

List of Illustrations

Chapter 1

Introduction

Music should make more sense once seen through listeners' minds.
– Marvin Minsky: "Music, Mind & Meaning"

Communication between musicians, verbal as well as musical, assumes certain shared concepts and experiences. Observing, for example, a rehearsal of chamber music, or a piano lesson, one might hear a comment such as, "Broaden the end of the phrase." Interpreting that instruction engages a complex collection of listening and performing skills, which must be related to each other in a reasonably precise way. However, the necessary relations are rarely described verbally beyond the use of just such admonitions. If a student were to shape the phrase poorly, a typical next response for the teacher would be simply to play, or sing it.

Pursuing such common musical effects in computer music systems often leads to alien and unwieldy constructions, precisely because the software does not share the concepts and experiences that underlie musical discourse. Many of the most persistent problems in computer music ("mechanical" sounding performances, lack of high-level editing tools) come from an algorithmic inability to locate salient structural chunks or describe their function. Though research has begun to show us systematic ways in which human performers add expression to their rendering of a score [Palmer88], a general application of the fruits of this research will be impossible before programs can find the appropriate structural units across which to apply expressive deformations. In other words, it does computer music systems little good to know how human players broaden phrase boundaries, if those systems cannot find the phrases in the first place. Among the concepts the machine would have to employ to "broaden the end of the phrase" are beat, harmonic progression, meter, and decelerando (Fig. 1). These concepts rely in turn, I maintain, on even more primitive perceptual features such as loudness, register, density, and articulation, and the way these change in time.

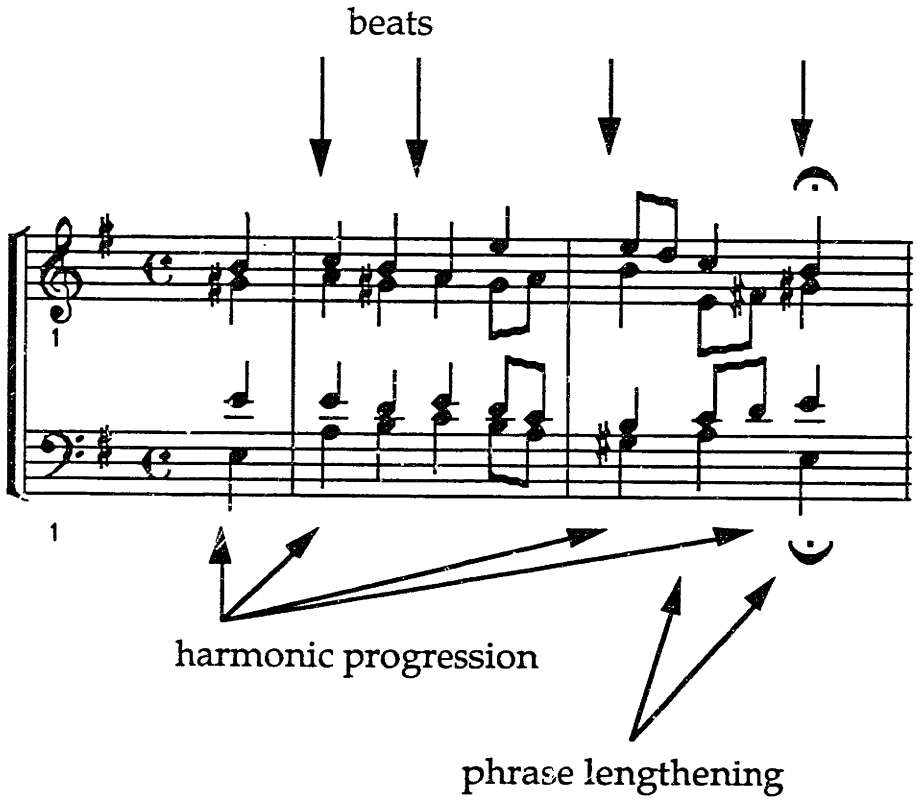


Fig. 1

This thesis describes *Cypher*[1], a computer program that can learn, recognize, and make use of structures such as those musicians commonly assume. Further, the program provides an interface which allows a user to indicate certain types of responses, to be generated when particular constructs are found. The program is able, by relying on musical concepts and their recognition, to make sense of and accomplish an instruction such as "broaden the end of the phrase." A user can develop, through a process of experimentation and refinement, various configurations of recognition and response, leading to a performance situation in which the computer can follow the evolution and articulation of musical ideas and contribute to these as they unfold.

---

[1] *Cypher* is an anagram of hyperC, a reference to the fact that the application is written in the C programming language, with some facilities added to handle scheduling and MIDI processing. In this respect the programming environment resembles Joseph Chung's HyperLisp, and this resemblance is obliquely noted in the name of the application.

1.1 Overview

Cypher is an interactive real-time system, able to use musical concepts in listening and composing. The implementation is based on a societal architecture, in which many small, relatively independent agents cooperate to realize complex behaviors. Contributions of this thesis come from the application of such an architecture to musical problems, achieving an unprecedented depth in real-time machine analysis, and sensitivity to context in composing a response. The collection of agents made, their organization into larger agencies through the cross-communication of results, and their mutual influence during processing, are the primary theoretic advances reported here. Specific accomplishments of the system demonstrate the power of the theory; *Cypher* is able to perform the following tasks, simultaneously, in real time:

- Classify several perceptual features arising from a musical context, and attach compositional responses to them
- Track the change in behavior of these features over time
- Identify the central pitch and mode of local and higher-level harmonic areas, such that chord function can be tracked and used compositionally
- Find the likely beat period of incoming musical data, coordinating the analysis with harmonic and featural classifications
- Change the nature of responses made to a musical context according to a set of production rules implementing the program's aesthetic preferences
- Learn and look for new occurrences of patterns in the domains of harmonic progressions and melodic intervals
- Group incoming material into phrases, following discontinuities of features, harmonic activity, and beat placement
- Adjust program output to accord with the harmonic, rhythmic, and featural behavior of musical input completely unspecified in advance

This thesis demonstrates the power of considering as much musical context as possible in addressing specific analytic tasks, such as beat tracking or harmonic identification. Further, descriptions of context on many levels enable the generation of another musical voice showing novel behavior, which still is recognizably related to rhythmic and harmonic activity in the

source material. The problems this implementation encounters arise when the program is not able to consider enough context: expanding the analysis down to the audio (timbral) level, and up to levels above those already present, would enable even more powerful musical descriptions and composition.

The work described here is an applied music theory: ideas about the description and generation of music are formalized to the point that they can be implemented in a computer program, and tested in real time. In the areas of harmonic and rhythmic analysis, and algorithmic composition techniques able to adapt to an ongoing musical context, *Cypher* represents a completely specified method whose capacities and limitations can be observed during interaction with complete musical examples. Though textual traces of program activity can provide a detailed account of the processing performed, the application of theoretical ideas in a real-time context brings the intellectual enterprise to a point of contact with real musical examples where the ear can judge the success of the theory.

An attempt was made to fit the facilities outlined here with a learning procedure able to find and retain meaningful musical constructs. The criterion for retention in the learning scheme was frequency of repetition, with harmonic, melodic, or rhythmic patterns stored in a long-term memory if they appeared repeatedly in musical examples. An implementation of this idea was written, and is described in Chapter 6. That experience has shown that frequency of repetition alone is an insufficient heuristic for successful learning. Criteria intrinsic to the material being analyzed must be developed to decide which inputs merit even short-term retention as candidates for matching against possible reiterations, or as interesting enough in their own right to be used as source material for the program's output. Though a good learning technique cannot fairly be claimed among the achievements of this thesis, I believe the analysis processes which have been developed offer a firm foundation for the expanded heuristics I suggest, and will review these possibilities in an evaluation of the pattern processing described in Chapter 6.

1.2 Music Cognition

The practice and pedagogy of music often are overly compartmentalized. Surely the skills involved in composition, performance, listening, and analysis overlap to a much greater degree than is indicated by their course curricula. Probably the skill on which all these disciplines rely more than any other is listening; "having a good ear" is said of skilled musicians of any specialty. The contrast between listening and analysis should be drawn here: the act of analysis is related to listening, as are all musical skills, but differs in two ways relevant to this discussion. First, music analysis has random access to the material – the analyst proceeds with the text of a written score, which he or she can consult in any order, regardless of the temporal presentation of the piece in performance. Second, the analyst is often concerned with learning the compositional methods used in constructing the piece [Cook87]; depending on the method, these may have anywhere from a great deal, to almost no perceptual salience for the uninitiated listener. The listener, by contrast, is constrained to hear the piece from left to right, as it were, in real time. There is no random access – rather, the music must be processed as it arrives. The groupings and relations the listener forms can only arise from cognitively available perceptual events, and the interaction of these events with short- and long-term memory. Seen in this light, the problems of music listening are simply part of the larger problem of human cognition.

The remainder of this section will sketch the network of relations to areas of cognitive science, music theory, and artificial intelligence in which this thesis is embedded. The most pervasive influence should be mentioned first: the theory of intelligence and cognition put forward by Marvin Minsky in his book *The Society of Mind* [Minsky86]. The central idea of Minsky's theory is that the performance of complex tasks, which require sophisticated forms of intelligence, is actually accomplished through the coordinated action and cross-connected communication of many small, relatively unintelligent, self-contained *agents*. The impact of this idea on the construction of *Cypher* will be seen at every turn, and described in detail as we proceed.

1.2.1 Relation to Human Listening

A primary goal of this thesis has been to fashion a computer program able to listen to music. The inspiration and touchstone for successful music listening is, naturally, human performance of the task, and *Cypher* implements what I take to be a method with plausible relations to human music processing. Still, I do not make the stronger claim that this program is a simulation of what goes on in the mind. For one thing, we simply do not know how humans process music. Cognitive science is a collection of disciplines moving toward theories of human information processing [Posner89], and music cognition is the branch of that field devoted specifically to aspects of human intelligence as they apply to music [Sloboda85].

Though we are as yet unable to verify these theories completely, due to the elusive nature of the thought processes they seek to explain, some lines of thought in cognitive science as a whole, and music cognition in particular, have influenced the development of this thesis greatly. However, it is irrelevant to the force of this work whether or not the processes implemented in it correspond directly to those at work in the human mind. At best, the program offers an existence proof that the tasks it accomplishes can be done this way. The point, in any case, is not to "reverse engineer" human listening, but rather to capture enough musicianship, and enough ways of learning more complete musical knowledge, to allow a computer performer to interact with human players on their terms, instead of forcing an ensemble to adapt to the more limited capabilities of an unstructured mechanical rendition.

The work in music cognition which has most deeply influenced the development of this thesis concerns the elicitation of structure from an ongoing musical flow, and the ways such structuring affects the comprehension, performance, and composition of music. Examples of these forebears include the schemata theory of musical expectation [McAdams87], the influence of structural understanding on expressive performance [Clarke89], and various formal representations of pitch and rhythmic information [Krumhansl90].

The field of music cognition is relatively new, particularly in comparison with music theory – a centuries-old collection of techniques for describing the construction of musical works, and the ways they are experienced. Music theory has long dealt with the tendency of Western music to exhibit directed motion, or dramatic form, in its progression through time. Observers such as Wallace Berry and Leonard Meyer have discussed goal-oriented patterns of tensions and relaxation [Berry76], and the way these arouse expectations, which are then confirmed or denied, in the mind of the listener [Meyer56]. The research in music cognition cited above is clearly an outgrowth of such lines of thought; similarly, this thesis is deeply rooted in those music theories that describe musical discourse as movements of progression and recession around points of culmination and fulfillment. In particular, the idea that enculturation and gradually acquired expectations of typical continuation play a major role in music listening has shaped much of the design of this computer program.

Expectation is, to be sure, only a part of what goes on during listening; for many kinds of music, not even the most important part. I am convinced that listening to music engages all of our cognitive capacity in a deep, significant way, and that this stimulation is what makes music a universal and life-long interest for every human culture. This thesis does not, of course, capture all of that richness. Rather, the focus here is on developing a computer program that can learn to recognize some facets of Western musical style, and incorporate that knowledge in an evolving strategy for participating in live musical contexts.

In the remainder of this text, I will use words such as "understanding", "intelligence", "learning", and "recognition" to speak of program function. It is not the point of this discussion to further fan the flames of debate raging around the use of these words to describe the behavior of a computer program [Winograd&Flores86]. The work described here actually exists; it has been used with human musicians in a wide variety of settings over a period of three years. The program is said to "understand", or exhibit "intelligence", in the sense that musicians interacting with *Cypher* feel that it is behaving musically, that they can understand what it does, and can see a musically commonsensical relation between their performance and the program's. That

is the level on which this work is meant to function, and "understanding" is used to describe a quality of that functionality as it is experienced by musicians interacting with the program.

1.2.2 Relation to Human Composition

The *Cypher* listener implements a particular approach to segmenting and classifying MIDI representations of musical streams. The analysis is always performed in the same way, and the messages that the listener may send are fixed. On the composition side, however, the kinds of processing available vary much more widely. One of the goals in fashioning *Cypher* was to make a framework on which many different compositional styles could be hung. The design of the program requires a user to configure the composition section to respond to messages emanating from the listener. In the current implementation, a diverse array of compositional methods is provided, with which the user may construct responses. In an ideal implementation, this array would be supplemented with a sophisticated music programming environment, allowing each user expand the collection of composition methods to include his or her own techniques. Indeed, as we shall see, the listening part of *Cypher* has already been isolated and built into an object for the Max programming environment [Puckette86], allowing users to program their own responses to listener messages.

My assumption is that the *Cypher* listener implements a level of musical competence able to navigate many common Western styles: much of this thesis is devoted to substantiating that claim. The methods provided by the composition section cannot be exhaustive, but have proven equal to the demands of many different musical contexts, different musicians, and, certainly, to the interests of my own compositions made using the program. The composition methods implemented arose from two intersecting sets of demands: 1) the composition of original works of music, reflecting my own musical thought and technique, and 2) experiments with different styles, meant to test and demonstrate the capacity of the program to function in musical contexts significantly different from my own usage.

Though the user is given primary control over which compositional methods the program will employ, *Cypher* is not entirely powerless to affect the outcome. An innovation of this thesis is to include a compositional *critic*, a process that examines the output of the composition section before it is sounded, evaluates it, and subjects the output to modification if the critic's aesthetic guidelines are not met. The critic's evaluation process, and modification techniques, use concepts of regularity, direction, and rate of change. It is in this sense that the composition section approaches the cognitive processes of human composers.

The composition methods are a contingent, idiosyncratic collection, prompted by the considerations mentioned above. The way the critic alters the combination of methods in use, or affects the behavior of the methods themselves, is a consequence of perceptions and preferences built into the critic, which are expressed in terms of rates and direction of change. To express aesthetic criteria in these terms holds the critic to an image of music as motions of progression and recession around goal points, an image that is shared by the *Cypher* listener. While this image cannot be termed universal, it does bear strong relations to the concerns of many human composers of Western art music.

1.2.3 Relation to Human Performance

The performance of music is determined by the performer's understanding of the composition he or she is playing. Performance pedagogy on all levels emphasizes the skill of interpretation, transcending the mechanics of pure technique. Once a performer has a conception of the music to convey, all of the specifics of physical interaction with an instrument are executed in accordance with the expression of that conception [Clarke87]. One of the most active research areas in music cognition explores the structure of a performer's musical concept, and how such structures can account for the deviations observed between a human player's expressive performance, and a literal, mechanical rendition of the same composition [Palmer88].

This thesis takes the first steps toward achieving expressive performance with a computer program. The structure *Cypher* derives from a musical

performance corresponds in a strong sense with the concepts human players use to shape their execution. Recognition alone, however, is not enough. To perform expressively, the player (human or machine) must be looking ahead, anticipating the future motion of the musical discourse. A program performing expressively, therefore, must have knowledge of typical continuations in a style, if it is to shape its performance in accordance with the anticipated shape of some musical structure.

The representations and learning capabilities of *Cypher* present a prototype of the facilities necessary for remembering and recognizing typical musical situations. If aspects of some directed motion can be recognized, an expected continuation can be asserted, and expression of the current performance adjusted to anticipate and emphasize the arrival of the motion's goal. By continually analyzing the music played to it, remembering patterns which have been repeatedly presented to such a degree that there is a reasonable expectation they may be encountered again, and matching new input against the repertoire of known progressions, the program can improve its knowledge of musical styles, and its capacity for performing expressively within them.

1.3 Interactive Music Systems

The program described in this thesis is an example of an interactive music system. Before we begin to examine *Cypher* in detail, we shall review the general area of interactive systems. First, we contrast them with the dominant method of computer music generation, one which arises from a conception of production related to the technology of tape recording and playback. Then we discuss the components and typical function of interactive music performance and composition systems.

1.3.1 Tape Recorder Paradigm

Though increasing numbers of interactive systems are being implemented and used in performance, the leading paradigm for computer music remains the tape recorder. Either sounds are produced for and stored on an actual tape recorder, or its functional analog, the sequencer, is used to record and edit

MIDI events, and play them back in a fixed order. In either case, the responsiveness and expression of live human performance are deadened, whether humans are actually playing along with the sequenced music or not. The advantage of the tape recorder is the ability it provides to shape and perfect musical gestures to the satisfaction of the composer. The rise of the home audio system, and the custom of listening to recordings of music, have provided a natural outlet for the productions of such studio work. In a live performance situation, however, the tape recorder is the wrong paradigm. For the listener, attending concerts of tape recordings has not proven to be a riveting experience, and for the performer, playing with a tape recorder is as little satisfying as practicing with a metronome.

Interactive systems reject the tape recorder paradigm in favor of an idea of extended musical instruments [Machover&Chung89], or programs emulating the example of human performers and improvisers [Lewis85]. In either case, the very idea of interaction demands that the program be able to track a live performance, and change its own behavior in response to musical developments. Though a wide variety of problems are addressed by the implementation of interactive systems, making the program responsive to musical developments requires the transfer of some degree of musicianship to the machine. The musical understanding implemented, and the way that understanding is employed, defines the space of musical purposes the program can usefully address. The program must be able to hear, at least; according to the aesthetic agenda, it may need to know composition as well.

Western music composition has known for centuries the power of using processes to generate musical material and forms. Music theory, in some of its manifestations, is the scholarly attempt to describe the process of listening to music. Computer systems able to implement this work in real time allow the musician to assess the validity of the intellectual enterprise by hearing its function in live musical contexts. The construction of formal processes is judged by the ear, and sound, not through more words, and paper.

1.3.2 Components of Interactive Systems

Most current interactive systems make use of the Musical Instrument Digital Interface (MIDI) standard, a hardware specification and communications protocol which allow computers, controllers, and synthesis gear to pass information between them [Loy85]. MIDI abstracts away from the acoustic signal level of music, up to a representation based on the concept of notes, comprising a pitch and velocity, which go on and off. Though the standard has had the effect of greatly expanding research into interaction, the loss of timbral information is one of several limitations adoption of MIDI tends to impose [Moore88]. The rest of this thesis will assume use of the MIDI standard. The limitations, though keenly felt, are still outweighed by the speed, facility, and concentration of attention on the control level which MIDI makes possible.

In a typical interactive system, a MIDI instrument (such as a keyboard, drum controller, pitch-to-MIDI converter, etc.) sends data to a computer, which has been equipped with a MIDI interface. The computer's program reads and processes the incoming MIDI data, and sends out commands of its own through the same interface to synthesizers, samplers, or other MIDI output devices. The processing and response take place in real time; the program changes its behavior in relation to information arriving from musical input devices, and can react to this information quickly enough to take part in a live musical performance.

Consider a machine listener. The task is to examine an incoming stream of MIDI data and extract useful information from it, such as recognizing the style, or identifying the harmonic structure, or finding the beat. Identifying harmonic structure, for example, requires reading and analyzing successive pitches to detect chord changes and derive key signatures [Scarborough et al.89]. Finding the beat demands analyzing note onsets to notice those regularly spaced attacks which could be interpreted as representing a beat-like periodicity [Chung89; Rosenthal89]. Such analyses all must be able perform certain basic functions: to quickly break down MIDI streams into manageable chunks; perform calculations on the data to extract the desired features; manifest the results of the analysis in some way; and equitably distribute

processor resources between parsing, analysis, and output. If the program is to react, through MIDI, according to the results of its analysis, further computations including the formatting and transmission of MIDI commands must successfully share processor resources with the rest.

Intelligent Music's *M* and *Jam Factory* [Zicarelli87], probably the most widely known instances of interactive music programs, provide a ready illustration of both the virtues and limitations of the field. These programs have graphic control panels, which allow access to the values of global variables affecting the musical output of the programs. Manipulating the graphic controls affects the nature of the musical output immediately. Selecting a different distribution function, for example, audibly affects what is heard: the user's ear becomes the primary arbiter of values for musically relevant variables.

At the same time, these programs necessarily embody far-reaching compositional choices which reflect the musical preferences of the authors. The same observation can be made concerning any interactive composition system, for the simple reason that composers, and the programmers of interactive systems, think about music in many different ways. It is at the same time the curse and the blessing of the field that interesting contributions tend to arise from composer/programmers, or composer-programmer collaborations, rather than any single general-purpose tool: the curse because wheels are reinvented many times over, together with the blessing of an abundance of approaches and solutions. *Cypher*, no less than other efforts, harbors its own brand of parochialism. But the musicianship captured by its ways of listening, and its ability to learn, remember, and use new musical knowledge, break new ground in making computers worthy partners of human musicians on the concert stage.

Chapter 2

Cypher – General Design

Cypher is an interactive computer program for composing and performing music. The program has two main components: a listener and a player. The listener (or analysis section) characterizes performances represented by streams of MIDI data, which could be coming from a human performer, another computer program, or even *Cypher* itself. The player (or composition section) generates and plays musical material.

There are no stored scores associated with the program; the listener analyzes, groups, and classifies input events as they arrive in real time without matching against any preregistered representation of any particular piece of music. The player uses various algorithmic styles to produce a musical response. Some of these styles may use small sequenced fragments, but there is never any playback of complete musical sections which have been stored in advance for retrieval in performance.

In the course of putting together a piece of music, most composers will move through various stages of work – from sketches and fragments, to combinations of these, to scores which may range from carefully notated performance instructions to more indeterminate descriptions of desired musical behavior. Particularly in the case of pieces which include improvisation, the work of forming the composition does not end with the production of a score. Sensitivity to the direction and balance of the music will lead performers to make critical decisions about shaping the piece as it is being played.

Cypher is a software tool which can contribute to each of these various stages of the compositional and performance process. In particular, it supports a style of work in which a composer can try out general ideas in a studio setting, and continually refine these toward a more precise specification suitable for use in stage performance. Such a specification need not be a completely notated score: decisions can be deferred to the performance, to the point of using the program for real-time improvisation. The most fundamental

design criterion is that at any time, one should be able to play music to the system and have it do something reasonable; in other words, the program should always be able to generate a plausible complement to what it is hearing.

2.1 Connections between Listener and Player

Let us sketch the architecture of the program: The listener is set to track events arriving on some MIDI channel. Several perceptual features of each event are analyzed, and classified. These features are density, speed, loudness, register, duration, and harmony. On this lowest level of analysis, the program asserts that all input events occupy one point in a *featurespace* of possible classifications. The dimensions of this conceptual space correspond to the features extracted: one point in the featurespace, for example, would be occupied by high, fast, loud, staccato, C major chords. Higher levels look at the behavior of these features over time. The listener continually analyzes incoming data, and sends messages to the player describing what it hears.

The user's task in working with *Cypher* is to configure the ways in which the player will respond to those messages. The player has a number of methods of response, such as playing sequences, or initiating compositional algorithms. The most commonly used method generates output by applying transformations to the input events. These transformations are usually small, simple operations such as acceleration, inversion, delay, or transposition. Though any single operation produces a clearly and simply related change in the input, combinations of them result in more complicated musical consequences.

Specific series of operations are performed on any given input event according to connections made by the user between features and transformations. Establishing such a connection indicates that whenever a feature is extracted from an event, the transformation to which it is connected should be applied to the event, and the result of the operation sent to the synthesizers. Similarly, connections can be made on higher levels between listener reports of phrase-length behavior, and player methods which affect the generation of material through groups of several events.

2.2 Hierarchies and Networks

Cypher's listener and player are both hierarchical structures: the listener strives to understand a musical stream on several levels, and the player executes procedures on different levels to create music. The levels of these hierarchies are distinguished in three ways. First, higher levels refer to collections of the objects treated by lower levels. For example, on the listening side, the lowest level examines individual events, while the next highest level looks at the behavior within a group of such events. Second, higher levels use the abstractions produced by lower levels in their processing. So, the second level listening agents, which describe groups of events, will use the classifications of those events made by a lower level analysis to generate a description. Third, because of the temporal nature of music, groups of events will be extended through time: therefore, higher levels in the hierarchy will describe structures which span longer durations of time.

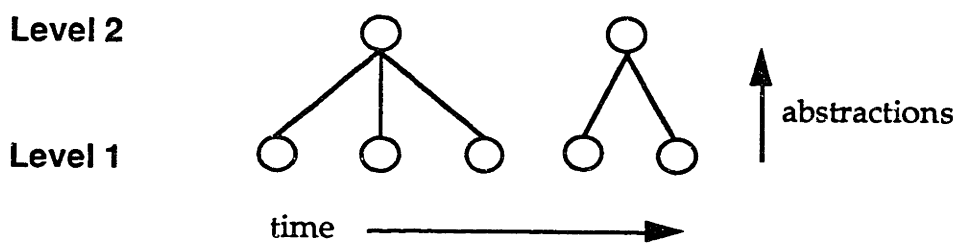


Fig. 2

Figure 2 shows some important ways relations are drawn between the events analyzed and generated by the program; however, this way of looking at the information is only one of many perspectives *Cypher* adopts in the course of carrying out various tasks. This first perspective is so strictly hierarchical that we can best depict it as a tree structure; sub-events are connected to only one super-event, and not to each other, etc. Other perspectives are not related in such an orderly way. This section will review the organizations suggested by different perspectives on the data under consideration, and by the kinds of

connection and communication linking the agents that process it. Further, we will relate the discussion to other representational schools in music theory.

2.2.1 The Progressive Perspective

We are considering various perspectives on the operation of *Cypher*. One important axis around which to organize these perspectives separates the raw material from the processing. Some structures concern the way the sounding events, which make up the fabric of the analyzed or generated textures, are grouped and related. The processes which perform the analysis and generation are themselves operative on different levels, and have their own connections of communication and grouping. We have already seen one perspective on the musical events, captured by the tree structure drawn above. Another perspective places more emphasis on their progression through time, and the associated relations of succession and precedence.

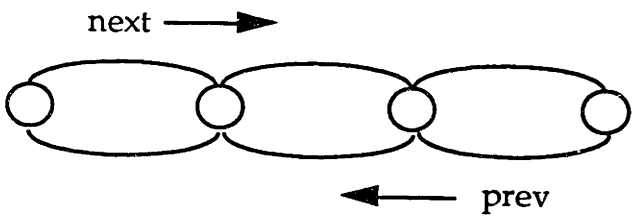


Fig. 3

Figure 3 illustrates the progressive perspective. The objects shown represent the same sounding events pictured on level 1 of the previous figure. There the emphasis was on their subsumption in meta-events; here we see that pointers connect events to their neighbors in both temporal directions. By traversing the pointers, we can arrive at proximate events either earlier or later in time. The progressive perspective is adopted on other structures as well; harmonic progressions, patterns of rhythm or melody, and higher level groups of the *Cypher* events shown in Figure 3 all are related, at times, by the operations of succession and precedence.

Already we can see that multiple perspectives tangle the depiction of relations between sounding objects; we can perhaps combine progression and hierarchy into a single visual representation that makes sense:

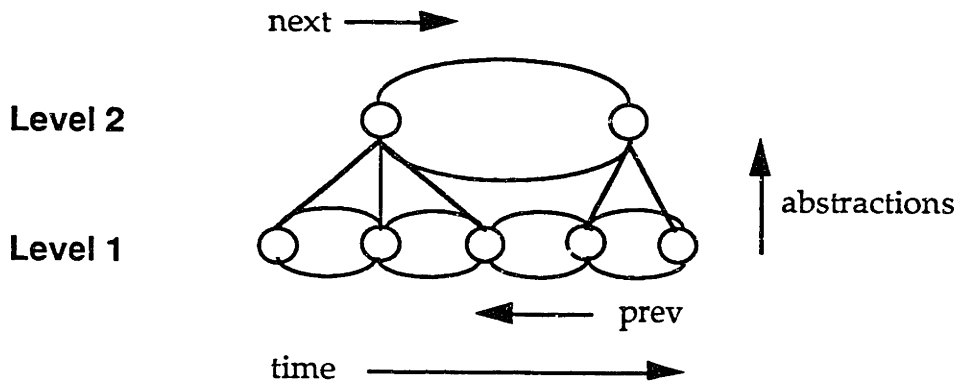


Fig. 4

All events are connected in a hierarchy, and simultaneously tied together in relations of succession and precedence. The addition of the progressive perspective takes our example far from the usual definition of a tree structure, however. Though it is clear enough to follow the combination of two relations, the effect of multiplying perspectives can be easily recognized as well: universal laws of relation between objects become obscured and complicated when there are several ways to connect and relate them.

In *Cypher*, a single, simple, universal form for relating objects, such as a tree structure, has been abandoned in favor of collections of ways to relate things. Our structures may become difficult to draw, but we will not be constrained by the representation to adopt a complicated solution to a problem that can be simply treated by a more appropriate perspective. Such pragmatism may seem straightforward, but as our review of other structural approaches will show in a moment, it is in fact an exception to common practice. Many prominent music theories, for example, devise a single structural perspective within which to describe musical behavior, and continue to adhere to that perspective no matter what difficulties are encountered trying to deal with the fullness of musical experience.

2.2.2 Message Routes

On the other side of the events/processing axis, *Cypher* comprises two large collections of interacting agents: those associated with the listener, and those with the player. The processes within each collection communicate with each other, and other kinds of messages are passed between the two collections. The hierarchical perspective on these processes is largely a function of the level of the events they treat: level 1 processes deal with the individual sounding objects seen in our earlier examples; level 2 processes deal with groups of these events.

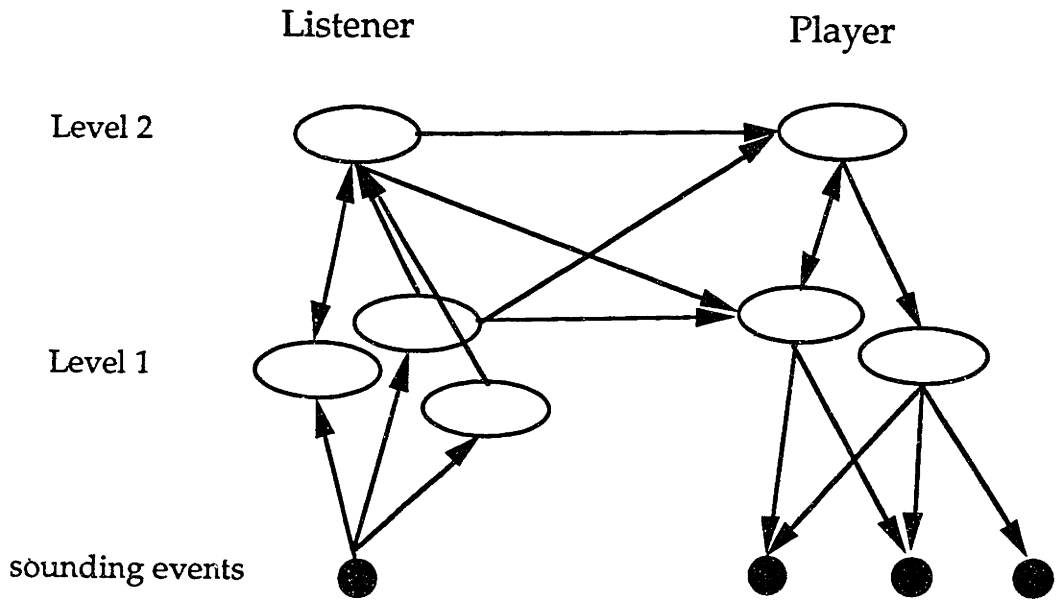


Fig. 5

Processes communicate by passing messages. These communication links form another perspective from which the ongoing musical information is regarded; the pattern of links, and the messages passing across them, give an indication of which features of the music are important for which tasks, and how the agents carrying out various tasks collaborate in their execution. Figure 5 shows some typical communication links within and between the two sides.

Though hierarchical, in the sense that there is a meaningful distinction to be made between levels of processing, we see that the lines of communication resemble a network of relations, rather than a more strictly formed tree structure. There are many processes on each side dealing with any single sounding event. Further, some general directions of the information flow can be noted: first, information tends to go up from the sounding events to analytical processes on the listening side, and on the playing side, generation methods are sending increasingly precise information down to the level of individual events, which, when complete, are sent on to the sound-making devices at the specified time. Another important regularity is that information passes only from the listener to the player, and not in the other direction. The only communication passing from the player back to the listener involves queries for additional data.

2.3 Perspectives from Music Theory

In this section, I will look at some important music theoretic uses of perspectives such as hierarchy and progression, comparing and relating the structures discussed to my own. The point of the discussion is to notice the descriptive power of each approach, whether this power can be amplified in combination with other structures, and what consequences the adoption of multiple perspectives has for the coherence and versatility of the theory as a whole.

2.3.1 Heinrich Schenker

The Galician music theorist Heinrich Schenker developed an analytic technique, in the early part of this century, with significant implications for the questions we are considering here. In particular, the combination of a strong concept of structural levels, together with the idea of progression from event to event within levels, points toward the progressive and hierarchical perspectives on *Cypher* events just reviewed.

> [Schenker] demonstrated that musical structure can be understood on three levels: foreground–middleground–background...Analysis is a continuous process of connecting and

integrating these three levels of musical perception. [Felix Salzer, in Schenker33 p. 14]

In the Schenkerian analysis shown below, we can clearly see the reduction of musical surface structures to a series of nested levels, with directed motion between certain events within each level.

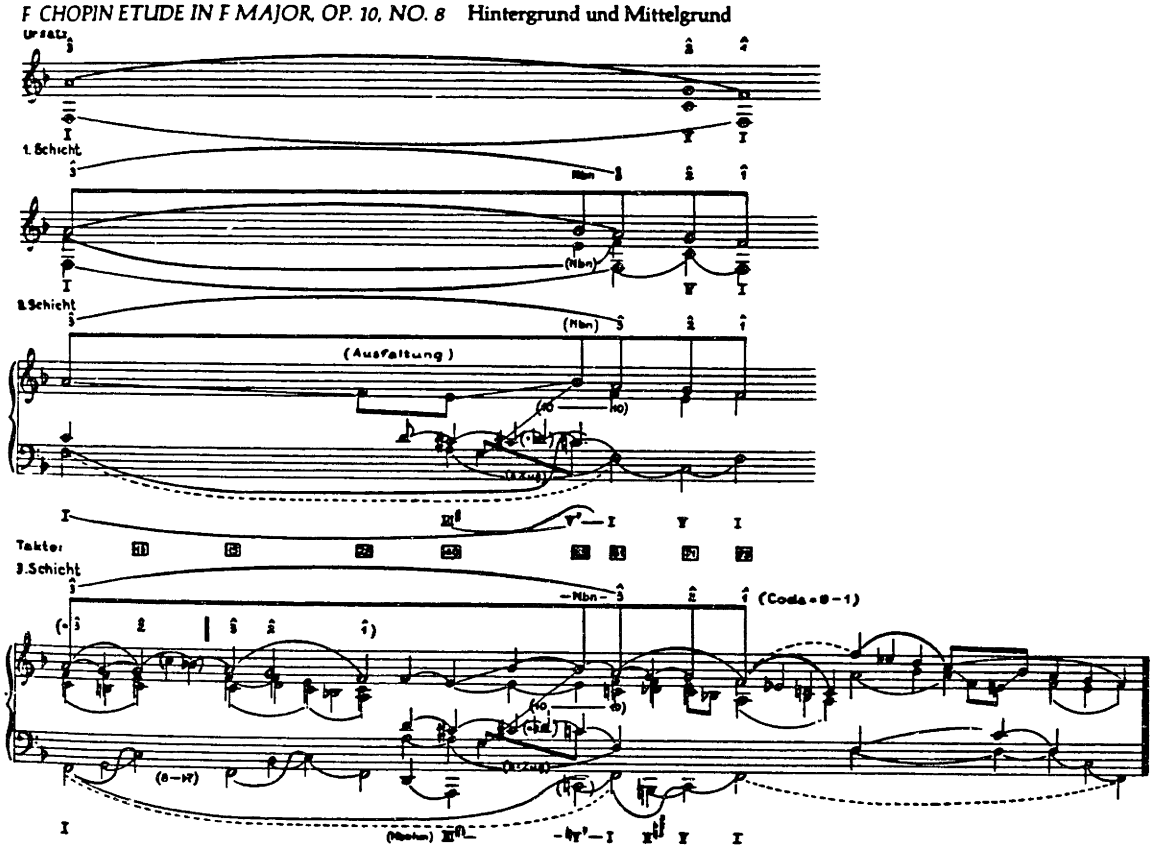


Fig. 6

Beyond the use of hierarchy and progression, another procedural contribution from Schenker's analysis is his use of *recursion*:

> Schenkerian analysis is in fact a kind of metaphor according to which a composition is seen as the large-scale embellishment of a simple underlying harmonic progression, or even as a massively-expanded cadence; a metaphor according to which the same analytical principles that apply to cadences in strict counterpoint can be applied, *mutatis mutandis*, to the large-scale structures of complete pieces. [Cook87 p. 36]

These three ideas – reduction, directed motion, and recursion – form a major part of the Schenkerian legacy to musical analysis. However, the application of the whole of Schenker's thought runs up against a rigidity of structure, which unnecessarily restricts the music it can successfully treat. The *Ursatz*, one of the foremost concepts associated with Schenker's name, is a background structure which Schenker claims is present in every well composed piece of music – even though, as Schenker was aware, the *Ursatz* clearly does not underlie a great percentage of the world's musical styles. The nonconformance of most non-Western music, indeed, of many centuries' worth of Western music, to the *Ursatz*, was a sign to Schenker that such music had not reached the summit of musical thought, dictated by the forces of nature, which he believed was achieved by Western tonal music in the classical style.[2]

Schenker's rejection of music not fitting his structural perspective is an extreme case, but not an isolated one. I find it to be emblematic of a recurring tendency in music theory to embrace a particular structural perspective so totally, that the theorist becomes blinded to the considerable power of a listener's mind to organize and make sense of music in ways unforeseen by any single theoretic account.

---

[2] "If, as I have demonstrated, all systems and scale formations which have been and are taught in the music and theories of various peoples were and are merely self-deceptions, why should I take seriously the Greeks' belief in the correctness of their prosody?" [Schenker35 p. 162]; see also [Narmour77 p. 38]

We may conclude by reiterating the procedural contributions of Schenkerian analysis: structural levels, progressive motion within levels, and recursion; and the cautionary tale of its inventor's application of them: that trying to enforce a particular perspective in describing musical thought can end by discarding a significant part of the phenomena the theory could well be used to explain.

2.3.2 Lerdahl & Jackendoff

The music theory of Heinrich Schenker, and the linguistic theory of Noam Chomsky, are two major influences on the work of composer Fred Lerdahl and linguist Ray Jackendoff, set forth in their book *A Generative Theory of Tonal Music* [Lerdahl&Jackendoff83]. Their theory is designed to produce representations of pieces of tonal music which correspond to the cognitive structure present in an experienced listener's mind after hearing the piece.[3] There are two kinds of rules in the theory: well-formedness rules, and preference rules. The first rule set generates a number of possible interpretations. The second rule set will choose, from among those possibilities, the interpretation most likely to be selected by an experienced listener.

One of the attractions of Lerdahl & Jackendoff's work is that it treats musical rhythm much more explicitly than do many music theories, Schenker's being an example. Another great strength is that it coordinates the contributions of several musical features, including meter, harmony, and rhythm. Further, predictions of phrase boundaries made by their model correspond well to reports from listeners under some test conditions [Palmer&Krumhansl87]. The output of the well-formedness rules is hierarchic and recursive, as Figure 7 shows. The tree structure at the top of the figure can be divided into hierarchic levels according to the branching structure; the rules generating branches and their relative dominance at each level are the same, and applied recursively.

---

[3] "[The theory] is not intended to enumerate what pieces are possible, but to specify a *structural description* for any tonal piece; that is, the structure that the experienced listener infers in his hearing of the piece." [Lerdahl&Jackendoff83 p. 6]

The progressive perspective, indicating directed motion within levels, is attenuated in the Lerdahl and Jackendoff version. The successive levels of the rhythmic interpretation shown in Figure 7 are related in an exacting tree structure; the only deviation from formal trees is that leaf nodes sometimes are linked to two adjacent nodes on the next highest level. The construction of these trees through application of the rule set makes possible comprehensive predictions of the strong/weak beat relationships experienced in a given musical context.

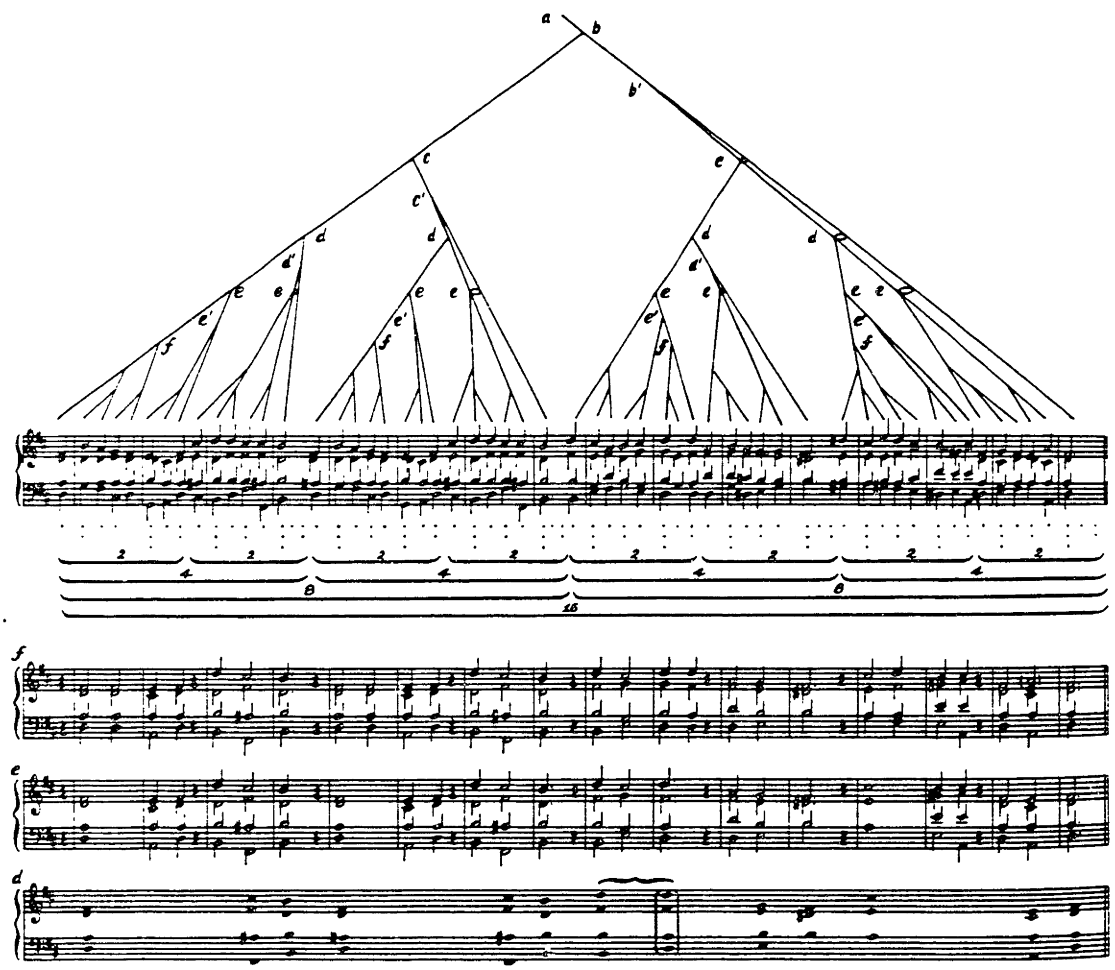


Fig. 7

Again in the case of the Lerdahl and Jackendoff theory, however, an exclusive reliance on one perspective leads to both uncomfortable accounts of cognition, and finally to proscriptive dismissal of music not conforming to the structure. One apparent problem with the full tree structure has to do with the cognitive reality of the upper reaches of the tree:

> Evidence for the highest level in this structure is rather sparse, and is confined to statements by a number of composers (Mozart, Beethoven, Hindemith) which indicate that they were able to hear (or imagine) their own compositions in a single 'glance'. [Clarke89 pp. 2-3]

Fred Lerdahl, in his essay "Cognitive Constraints on Compositional Systems", gives the following motivation for relating the theory elaborated in [Lerdahl&Jackendoff83] to compositional technique:

> Cognitive psychology has shown in recent decades that humans structure stimuli in certain ways rather than others. Comprehension takes place when the perceiver is able to assign a precise mental representation to what is perceived. Not all stimuli, however, facilitate the formation of a mental representation. Comprehension requires a degree of ecological fit between the stimulus and the mental capabilities of the perceiver. [Lerdahl88 p. 232]

Lerdahl complains that modern music composition has lost any connection between what he refers to as the compositional grammar used by a composer in constructing a piece, and the grammar used by listeners to structure it while listening. This argument leads to his "*Aesthetic Claim* 2: The best music arises from an alliance of a compositional grammar with the listening grammar." [Lerdahl88 p. 256] A recurring example in Lerdahl's essay is Boulez's composition *Le Marteau Sans Maître*, which does have, if one accepts Lerdahl's decomposition of the composing/performing/listening complex, a singularly striking decoupling of compositional and listening grammars. But, as Lerdahl points out,

> This account is complicated by the fact that, as noted above, Boulez created *Le Marteau* not only through serial procedures but through his own inner listening. In the process he followed constraints that, while operating on the sequence of events

> produced by the compositional grammar, utilized principles from the listening grammar. [Lerdahl88 p. 234]

In my view, Fred Lerdahl should not be so surprised that Boulez' serial technique, his "compositional grammar", is treated as if it were irrelevant. It *is* irrelevant. What makes *Le Marteau* a great piece of music comes from Boulez' musicianship, his "intuitive constraints". By concentrating on the "compositional grammar", to the point of staking aesthetic claims on it, Lerdahl is taking his eye off the ball. What is important is the way listeners make sense of music, a sense employed by composers, performers, and listeners alike. Lerdahl and Jackendoff's theory may well explain parts of that sense; they are the first to point out that their theory is not by any means complete. Basing aesthetic claims, and establishing constraints on composition, on an incomplete account, again amounts to overestimating the theory, and shortchanging the mind's capacity to deal with many different kinds of music.

2.3.3 Eugene Narmour

A forceful statement in music theory against an overreliance on tree structures can be found in [Narmour77]:

> If, however, as has been implied, the normal state of affairs in tonal music is *non*congruence of parameters between levels instead of congruence, it follows that analytical reductions should be conceptualized not as trees – except perhaps in the most simplistic kinds of music where each unit (form, prolongation, whatever) is highly closed – but as *networks*. That is, musical structures should not be analyzed as consisting of levels systematically stacked like blocks...but rather as intertwined, reticulated complexes – as integrated, nonuniform hierarchies.
>
> Unity would then be a result of the interlocking connections that occur when implications are realized between parts rather than as a result of relationships determined by the assumption of a preexisting whole. [Narmour77 pp. 97-98]

He observes in his article [Narmour84] that in fact much of what passes for hierarchical structuring in music theory is not hierarchic at all, but rather "systemic", by which he refers to "musical relationships which are conceived in Gestaltist fashion as parts of a completely integrated whole." [Narmour84 p. 138] These structures, he states, are not hierarchic because differentiations of material on lower levels are not reflected in their representation on levels higher up. Because of the loss of information as we travel up the tree, all individual characteristics of a particular piece of music are subordinated to a priori style traits; such analysis "reduces an idiostructural event to a default case of the style." [Narmour84 p. 135]

Narmour's own theory, the implication-realization model, has as a structural consequence the postulation of an extensive, multi-faceted network of connections between musical events on various levels of a composition. Following the work of Leonard Meyer and others, Narmour emphasizes the impact of a listener's expectations on their experience of a piece of music. Further, he recognizes the operation of multiple perspectives in music cognition:

> What makes the theory and analysis of music exceptionally difficult, I believe, is that pieces display both systematic and hierarchical tendencies *simultaneously*. And, as we shall see, this suggests that both "tree" and "network" structures may be present in the same patterning. [Narmour77 p. 102]

Eugene Narmour's theory tends to assume the goal-directed, expectation-based model of music cognition. It relies heavily on the ideas of hierarchy and progression; in fact, is the most consistently progression-oriented theory of the three I have reviewed here. Because Narmour sees progressions operating between non-congruent elements, however, his analytical structures tend not to resemble trees; for much the same reason, they are not recursive.

2.4 Level 1

The preceding sections sketched some of the perspectives on events and processing employed by various parts of *Cypher*. In particular, we saw precedents for some of the most common perspectives used (hierarchy,

progression, and networks) and some of the motivation for using multiple perspectives in dealing with musical contexts. In the remainder of this chapter, we will look at two levels of operation in *Cypher*. These are called level 1 and level 2; the numbering starts with 1, because I assume a level 0, which the program consciously avoids. Level 0 would correspond to the acoustic signal level, and treat things like the timbral quality of a musical context. By accepting the MIDI boundary between signal and control levels, *Cypher* defers treatment of level 0 to the future.

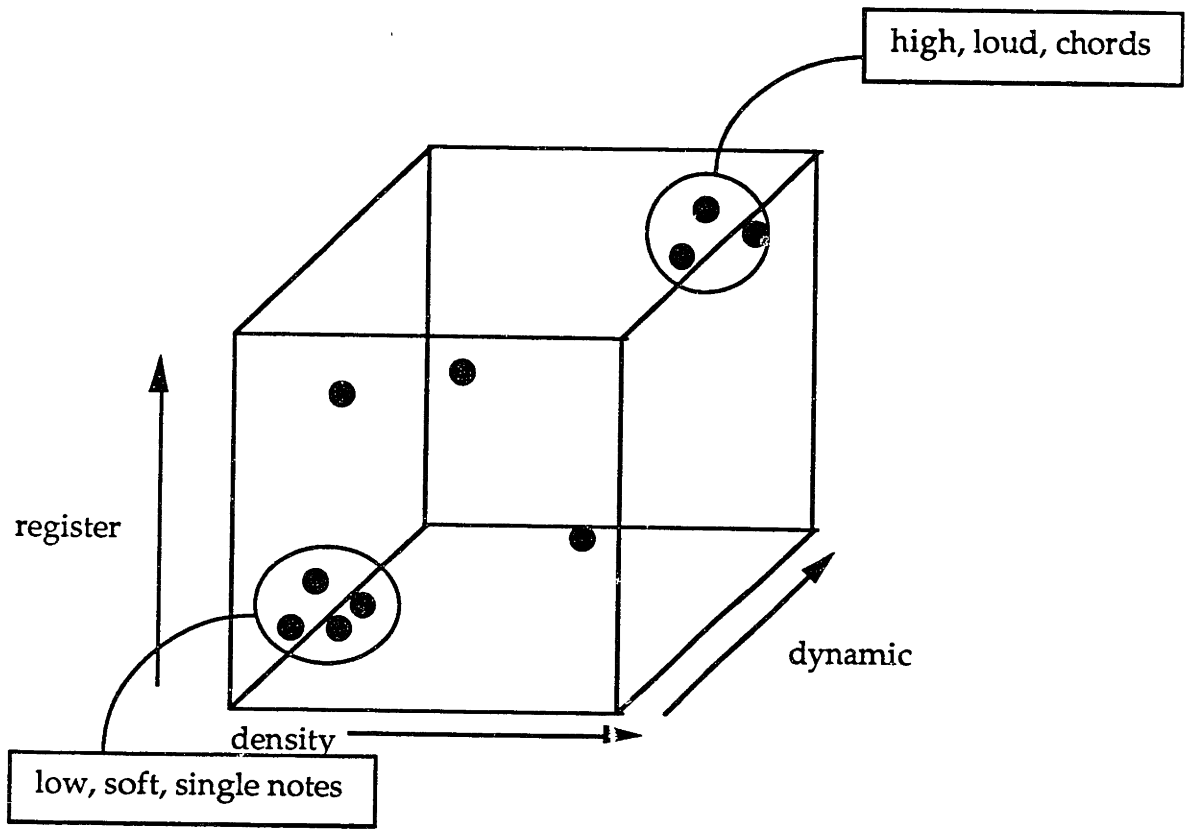
Cypher's level 1 comprises processes on the listening and playing side which deal directly with sounding events. Two more perspectives will be introduced: in the listener, level 1 processes collaborate to describe sounding events in terms of a *featurespace*, placing them in a conceptual area bounded by several dimensions corresponding to perceptual qualities. On the composition side, the output of level 1 can be regarded as the result of a series of *transformations* on some initial musical material.

2.4.1 Analysis

The listener process on level 1 classifies each incoming *Cypher* event as occupying one point in a multi-dimensional *featurespace*. This is a conceptual space whose dimensions correspond to several perceptual categories: placement of an object in the space asserts that the object has the combination of qualities indicated by its relation to the various featural dimensions.

To illustrate the concept, consider a three-dimensional featurespace, bounded by the perceptual categories register, density, and dynamic. Points occupying such a featurespace are shown in Figure 8. The position of the points in this example space are determined by assigning each event values for the three categories bounding the space; those values, taken as a vector, specify a unique location in the space for the event. In Figure 8, we see a cluster of points near the upper, right, back corner of the cube. These points represent sounding events that have been classified as high, loud, chords: the values assigned to each of these qualities is near the maximum endpoint of the scale. Similarly, there is another cluster of points near the lower, left, front corner

of the cube which, since the values assigned to all qualities are near the minimum endpoint, correspond to low, soft, single notes. We can see that events will be grouped more closely as their perceptual qualities are more closely related; points near each other in the space represent events with many perceptual qualities in common.



Featurespace

Fig. 8

The featurespace actually used by the level 1 analysis has six dimensions : register, loudness, vertical density, attack speed, duration, and harmony. The precise meaning of each of these terms will be defined in Chapter 3. The values assigned to each feature are a function of the MIDI data associated with the input event, and of the points in time at which the event begins and ends. For example, the *loudness* dimension is decided by simply comparing the

event's MIDI velocity to a threshold: those velocities above the threshold are classified as loud, and those below it as soft.

Classifying loudness into one of two categories is of course unnaturally restrictive: there are certainly many more musically meaningful dynamic gradations than two. The point of this research, however, is to work on the combination and communication of many analysis agents. Therefore, the individual analysis processes are kept skeletally simple. Scaling the loudness analysis up to much finer gradations would change nothing in the design of the system, other than making it bigger. Because the upgrade is straightforward and not central to the thesis, it has not been implemented in the research version.

The featurespace classification combines the output of each feature agent into a single result; this result is one of the main messages passed from the level 1 listener to the player. The contribution of any single feature can be easily masked out of the whole. In this way, the player is continually informed of the behavior of each feature individually, and in combination. Further, higher listening levels will use the featurespace abstraction to characterize the development of each feature's behavior over time.

Conceiving of sounding events as points in a featurespace facilitates computations of proximity and grouping: points close to each other perceptually will occupy proximate featurespace positions. Further, featurespaces have received considerable attention in other domains, for such calculations as *conceptual clustering*. This technique analyzes featurespace populations, and groups events which are close to each other in the space, but distinctly separated from other clusters. I will review the possible application of such ideas to music in Chapter 7.

2.4.2 Composition

The composition section of *Cypher* is designed to accommodate three types of compositional method: 1) transformation of material arriving from some MIDI source, 2) algorithms which generate material themselves, in one of a

variety of textural and gestural styles, and 3) performance from a library of sequences.

A *sequence* is a list of stored MIDI events, which can range in duration from gestures of a few seconds, to entire compositions many minutes long. The facilities for using sequences in *Cypher*'s composition section are primitive; in particular, once a sequence is launched, it is played back as it was stored, with no opportunity for real-time modification of the tempo, or other performance parameters. Sequences are handled by a process which schedules one second's worth of events from every open sequence file each time it is called. This sequence scheduling routine is called at least once a second, so sequences of arbitrary complexity can be included in the output without disrupting other processing.

The second class of composition methods comprises algorithms which generate material from some basic stored elements, often using constrained random operations to spin out a distinctive gestural type from the seed elements. Each of these algorithms is designed to manufacture a recognizable musical texture: shifting chords, quickly moving elaborations of a single pitch, or tremoli moving consistently up in the frequency space are examples. Any one of these routines can be invoked with a number of performance parameters, to set such things as the duration within which to continue generation, a pitch range to be observed, the speed of presentation, etc. This class of methods corresponds most closely to what is commonly referred to as *algorithmic composition.*

But by far the most heavily used method in the composition section involves the *transformation* of material, accomplished through the chaining of many small, straightforward modules, each of which makes some small, consistent change to material it receives as input. The action of these modules is cumulative: if more than one is used, they are applied serially, with the output of one operation being passed on to the input of the next. Though the action of any module taken singly is simple and easy to follow, longer chains of transformations can build up material that is quite complex, though deterministically derived from its inputs.

In the simple example following, a source chord is first sent through the arpeggiation module, which separates the pitches, and then through the looper, which repeats the figure given it.

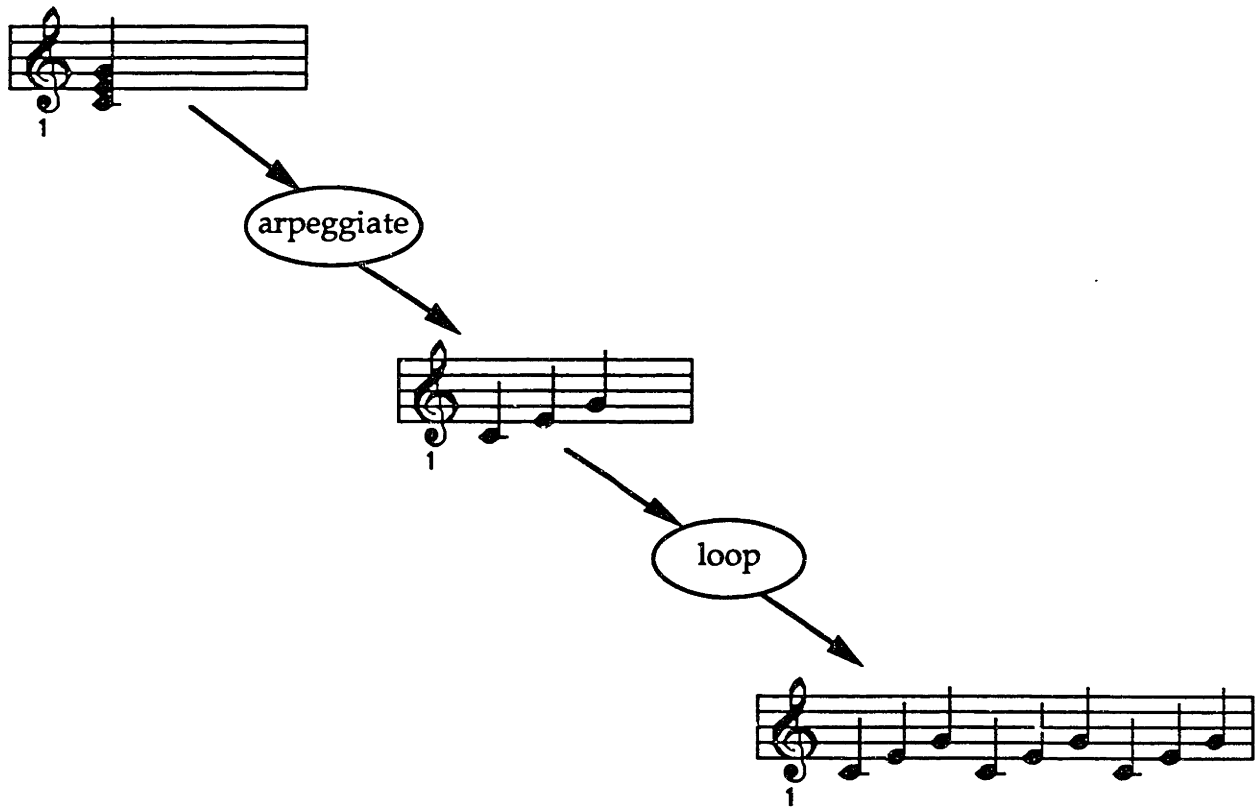


Fig. 9

2.5 Level 2

Cypher's level 2 is made up of listener and player processes which deal with collections of the events treated by level 1. On the listening side, level 2 looks at the featurespace classifications for each sounding event, and tries to find meaningful groupings. Within groups, level 2 notices patterns of regularity in the behavior of features.

On the playing side, level 2 processes change the operation of composition methods over groups of several sounding events. There are several ways to do this: making new connections on level 1 between features and

transformations, or breaking old ones; changing the behavior of level 1 transformation modules; executing specialized methods at group boundaries, or in anticipation of them.

2.5.1 Analysis

The listening processes on level 2 describe the behavior of several *Cypher* events. Level 2 processes examine the featurespace reports from level 1, and look for three main types of structure in the behavior of the features over time. First, one agent tries to group events together into phrases. A second agent looks at all events within a phrase, and decides whether each of the level 1 features is behaving regularly or irregularly within it. A third agent tries to determine, if a feature is changing, whether some direction can be detected in the pattern of change.

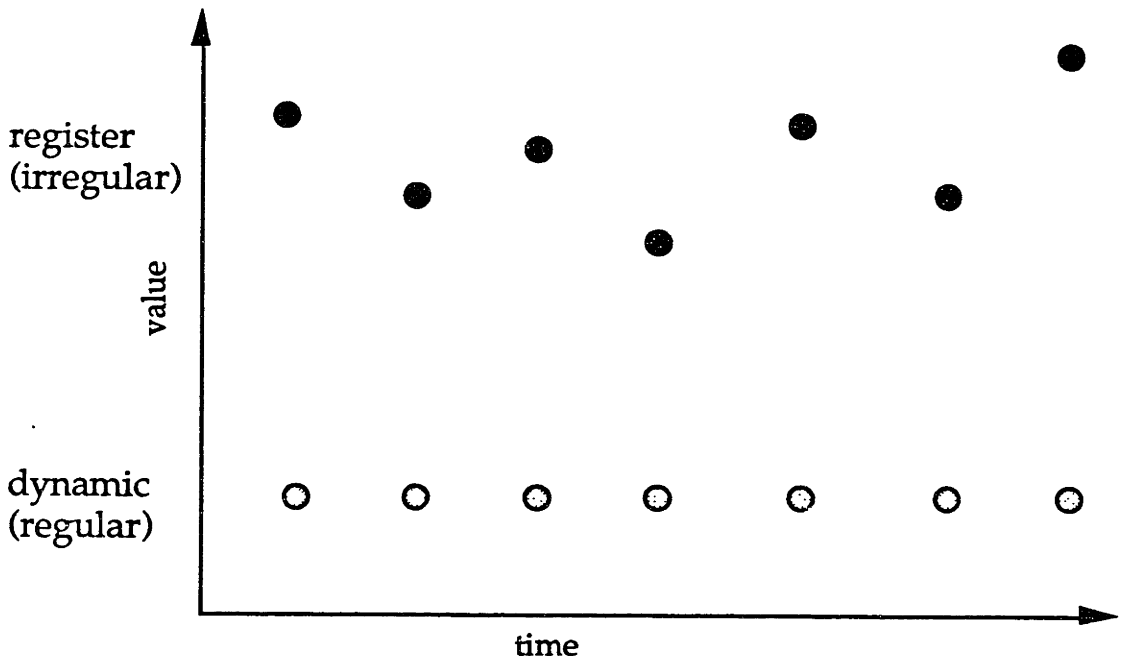


Fig. 10

In Figure 10, we see the feature classifications for several *Cypher* events, played out over time. The black dots represent the register classifications for each event, and the grey dots the dynamic values. In this case, registral

behavior would be termed regular by the level 2 listener, and dynamic behavior would be flagged as irregular.

These three tasks (grouping, observations of regularity and direction) correspond to a general view of music in which change (or the lack of it) and goal-directed motion form the fundamental axes around which musical experience revolves. Such a view is espoused, for example, in Wallace Berry's book *Structural Functions in Music*:

> ...by recurrent reference to interrelations among element-systems, reciprocal and analogical correspondences are indicated in which the actions of individual elements are seen to project expressive shapes of progressive, recessive, static, or erratic tendencies. Progressive and recessive (intensifying and resolving) processes are seen as basic to musical effect and experience. [Berry76 p. 2]

Not all music exhibits such directionality, and therefore does not comfortably bear description using such terms. However, that some music *is* about directionality and change, particularly much Western music, seems to me equally evident. *Cypher* is biased toward looking for goal-directed musical behavior; however, the listener will still have something meaningful to say about music which is not primarily goal-directed: in that case, useful analysis will be shoved down, as it were, to level 1. Classifications of individual features and local musical contexts will take precedence over longer-term descriptions of motion and grouping.

By now we have seen much evidence of the hierarchical and progressive perspectives at work in the operation of *Cypher*. However, the third major procedural component of the music theories reviewed earlier, recursion, is not employed. The levels of *Cypher* have their own concerns, and their own processes related to the goals for each level. That the processes employed for each level's tasks are different, means that recursive invocation of them would be of no use. This is another reason that relationships in *Cypher* tend to resemble networks more than trees: the regularity of generation implicit in recursive analysis, which often generates tree-like structures, is not appropriate to the goals of the program.

2.5.2 Composition

Level 2 composition processes are invoked by messages arriving from the level 2 listener, and affect the behavior of the composition section over phrase-length spans of time. The messages sent out from the listener have to do with grouping and regularity. The processes on the composition side are clustered around those types of messages, and effect various kinds of change to the musical flow. One common strategy on level 2 is to make and break connections between features and transformations on level 1. This is a good way to achieve the complement to some kind of behavior observed by the listener. Reports of regularity for some feature, for example, could be met by processes generating change in the same domain.

Another strategy is to mutate the level 1 transformation modules; control variables relevant to the transformation method can be changed according to the nature of the input being reported by the listener and the current state of the variable. For example, the accelerando module can be made to accelerate its input more or less than it already does as a function of the speed feature found by the listener, and the accelerando rate currently active.

Level 2 is also the appropriate place to perform processes of expressive variation, which extend across groups of sounding events. Variation in timing, or loudness, can be used to accentuate structural boundaries between events as they are produced. The ideas of second level composition are closely allied with the concerns of level 2 analysis: the regularity, or types of change, of collections of events; the grouping together of such collections; and the direction of regular change, are attributes to be generated in the music emanating from processes on the second level of the composition section.

2.6 Listening Streams

Any number of distinct player streams can be assigned their own *Cypher* listener. A listener is called with a new event, and a pointer to some stream's analysis history. A player stream can emanate from any MIDI source, such as that coming from a human performer, from another computer program, or

from *Cypher*'s own player. *Cypher*'s current architecture maintains two listeners, each with their own histories. One listener is tuned to MIDI input arriving from the outside world; the other is constantly monitoring the output of the composition section.

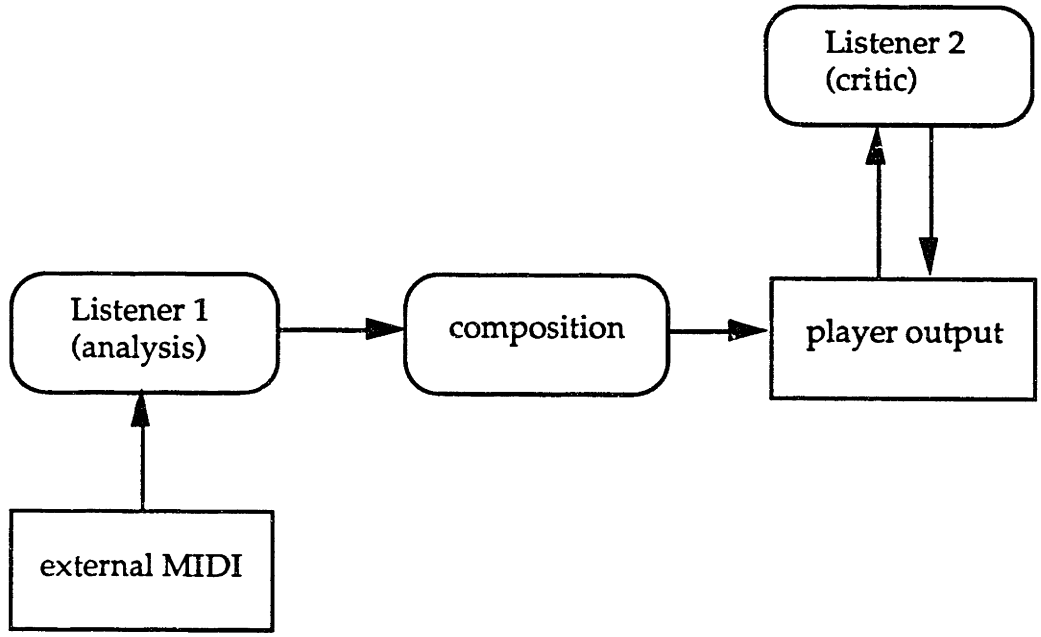


Fig. 11

We have just reviewed in some detail the process of analyzing a MIDI source, sending messages to the composition section, and producing novel musical material in response. The second listener shown in Fig. 11 represents a compositional "critic", another level of observation and compositional change, designed to monitor the output of the *Cypher* player, and apply modifications to the material generated by the composition methods before they are actually sent to the sound-making devices.

Interactive music systems typically have no way of evaluating their output – the composer/user of such systems steers the program in one direction or another without any way of measuring success other than by ear. One reason the problem has been so neglected is that it is treacherous; evaluating musical output can look like an arbitrary attempt to codify taste. I choose to look at it another way: though certainly arbitrary, developing a computer program with

a capacity for aesthetic decision-making is interesting in its own right, and for the quality of information which becomes available to the human developer in further evolving the program.

The information provided by the "critic" listener forms the foundation of a rule set governing the featural attributes, grouping, regularity, and direction of change in the output. The critic functions as a production system: a set of rules controls which changes will be made to a block of musical material exhibiting certain combinations of attributes. Tracing the analysis produced by the critic listener, and the changes introduced by the production system, an informed evaluation can be made both of the music being produced by the composition section, and of the effectiveness and stylistic traits of the critic.

2.7 Introspection

One easily detected, but nonetheless critical, condition for the program to notice about the music coming from the outside world is when the music has stopped. Among the possible causes of such a condition are that the piece is over, players are stopping a rehearsal to discuss the performance, the building is on fire, or that it is time for the computer to take a solo. *Cypher* is an enthusiastic soloist: it always takes the absence of other input as a signal that it should play more.

Since the most common generation method for the composition section involves transforming received input, the absence of any input would seem a crippling limitation. When performing alone, *Cypher*'s composer continues generation by transformation through the simple expedient of transforming its own output. This method of soloistic generation I call composition by *introspection*. Music is generated from a controlled feedback loop: the player produces some output, which is analyzed and characterized by a listener. The listener sends messages to the player about what it has heard (in other words, what the player just produced), and the player is instructed, through the connections between listener messages and composition methods, to transform its own previous output in some way.

There are two possible paths for the feedback loop just described: in the first, the critic simply takes over. Analysis of player output and any reordering of compositional methods occur in accordance with the production rules making up the critic's aesthetic decisions. The second path simply sends the output of the player back to the analysis listener, shown by the heavy black line in Figure 12 below:

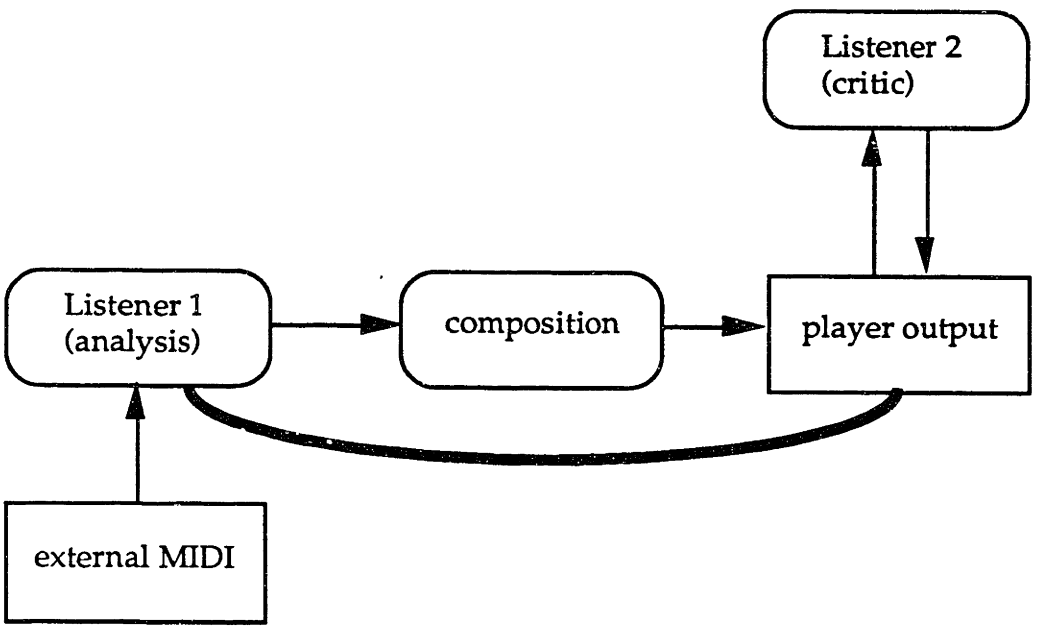


Fig. 12

The second feedback path provides a way for the user to control the connection of listener messages to composition methods. This facility provides another type of interaction with the program, allowing a human director, or a script of connection sets, to regulate the performance of *Cypher* during computer solos. Further, the output/analysis feedback path makes a good laboratory for refining production rule sets for later installation in the critic. A user can find composition methods appropriate to different configurations of analysis messages, and install these as the action parts of corresponding productions in the critic's rule base.

Applying the transformations to their own output turns out to be an interesting way to observe their effect. Feeding back on themselves, many

transformations lead to registral, temporal, or dynamic extremes, where they will remain until something disrupts the state. Using the connection mechanism to reorder the modules called by different configurations of the featurespace provides just such a disruption. Another approach is to mutate the low-level transformations when level 2 analysis finds features behaving regularly. Pinned behaviors are flagged as regular, and a subsequent mutation of the transformations, ordered by the level 2 player, sends output off in another direction.

Chapter 3

Listening

This chapter describes the operation of a *Cypher* listener, on all levels. Listening agencies will be discussed from the bottom up: beginning with individual feature classifiers, through the categoric analysis of harmony and rhythm, up to the second level organizations of regularity, grouping, and direction. As has been noted, the current implementation of *Cypher* maintains two listeners: one trained on the outside world, and another functioning as a critic of music generated by the program. At the end of this chapter, we will discuss ways in which the program could be expanded to track several players, and coordinate responses arising from the interaction of many analyses.

3.1 Feature Agents

The first stage of the listener trains a collection of feature classification agents on incoming *Cypher* events. These agents characterize various perceptual traits, including vertical density, horizontal density, register, loudness, and duration. Though the current implementation executes them in series, the feature agents could operate in parallel; there is, in this earliest stage, no communication between them. Earlier, I discussed the various perspectives adopted in considering events and processing in the listening part of *Cypher*, one of which was the hierarchical division of labor into various levels. The classification of features described here makes up the bulk of the listener's operation on level 1.

When all of the feature agents have finished their evaluations, these are combined into a *featurespace* word, a vector identifying one point in the space of event classifications, bounded by the features represented. Later stages of the listener use these featural reports, in conjunction with other processing, to perform higher-level tasks such as chord identification and beat tracking, and to characterize the evolution of feature classifications over time.

3.1.1 Featurespace Representation

The results of the feature classifications are stored in one bitvector of data, known as the *featurespace word*. Currently, the featurespace word is stored in 13 bits, as follows:

Feature	Precision	BitMask
density	2 bits	0 0000 0000 0011
register	2 bits	0 0000 0000 1100
speed	2 bits	0 0000 0011 0000
dynamic	1 bit	0 0000 0100 0000
duration	1 bit	0 0000 1000 0000
harmony	5 bits	1 1111 0000 0000

As I noted earlier, the restriction of these classifications to very few bits of precision is a convenience for the implementation of the research version. In any full-scale version, many of these features, with the possible exception of harmony, would need greater resolution. However, nothing of structural consequence would change – the packing and unpacking of data into and out of the representation would differ, and the feature agents themselves would have to be revised to take advantage of the increased precision. With these two modifications, the program could readily produce feature classifications with finer gradations.

3.1.2 Focus and Decay

Focus is a technique used to make feature judgements relative to the musical context actually heard. Rather than evaluating qualities against a constant scale, the measurement scale itself changes as new information arrives. When there is change within a very small range of values, the focus of the agent narrows to an area where change can be detected. When values are changing over a very wide field of possibilities, the focus pulls back to register change on a less refined scale.

Decay is the adjustment of the focal scale over time. The magnitude of the measurements made is relative to the range of change seen – the principle of focus. The scale against which measurements are made changes over time, as

data which is not reinforced tends to recede from influencing the current context – the principle of decay. The first featural agent we will consider, that classifying register, will provide a good example of these two techniques at work. The remaining agents make less initial use use of focus and decay: either their lack of precision, or more vexing theoretical problems, make them less amenable to such treatment. The relation of each agent to the concept, however, will be discussed as we proceed.

3.1.3 Register

The *register* agent classifies the pitch range within which an incoming event is placed. At the most basic level, this classification distinguishes high from low in pitch space, as these terms are universally understood. Making realistic registral judgements requires more precision than a separation of high from low, however, and is further sensitive to the conditioning effects of context and timbre. Our considerations in this thesis are outside the bounds of timbre, and our precision is limited to two bits. Still, registral judgements in *Cypher* are made against a scale derived from the pitch information actually experienced by the program.

One way to classify register would be to divide a piano keyboard up into some number of equally sized divisions, compare incoming events against these divisions, and assign classifications according to which division contained them. However, though splitting up the 88 keys of the keyboard will work well for piano music, and other ensemble textures which use most of the musical range, it makes little sense for other solo instruments, or combinations of instruments, using only a fraction of that span. Therefore, the register agent keeps a scale of low to high based on the pitches actually analyzed up to any given moment. As each new event arrives, it is compared to the scale stored for that event stream. If the event is lower or higher than either endpoint of the scale, the scale is adjusted outward to include the new event. For example, if the lowest point on the scale is MIDI pitch number 60, and a new event of 48 arrives, the low endpoint of the scale is changed to 48.

The precision of the classification reports is directly tied to the size of the scale against which the measurements are made. If the pitch scale for some stream

is less than two octaves, the register agent will only distinguish between low and high. Once the span opens out to over two octaves, possible classifications expand to four: low, medium low, medium high, and high. Consequently, judgements made about pitches played in a small ambitus will have fewer gradations than judgements concerning pitches presented in a more varied context. In the case of chords, the overall register classification is decided by a "majority" rule : the register with the most pitches of the chord is declared the register of the event as a whole. If more than one register has the same number of pitches in a chord, that is, if there is a tie in register classification after application of the "majority" rule, the lowest classification among those tied will be chosen.

The adjustment of the endpoints of a stream's pitch scale as new pitch information arrives is an example of focus in a feature agent. A necessary complement is the principle of decay, which also changes the endpoints of the stream's pitch scale over time, but in the opposite direction. These two operations together involve the register scale in a process of continual expansion and contraction. As we have seen, when new pitches arrive which exceed the bounds of the previously established range, the register scale grows by replacing one of the endpoints to match the new data. If an endpoint has not been reinforced for five seconds, that endpoint shrinks in towards the opposing endpoint by one half-step (one MIDI pitch number). Thereafter, the endpoint will continue to shrink inward by one half-step, every half second. When a pitch arrives to reinforce or extend an endpoint, the scale again grows outward to meet it, and the decay timer is reset to five seconds. After a new duration of five seconds with no additional information near the extreme, the scale will again begin to shrink inward.

3.1.4 Dynamic

The *dynamic* agent classifies the relative loudness of events. In this case, the nature of the feature demands careful consideration of the focus and decay techniques. For MIDI data, there is already given a highly significant scale of possible loudness values against which events can be compared: standard MIDI encodes velocity information in seven bits, giving a range of values varying from 0 to 127. Unfortunately, the perceived loudness of two different

synthesizer voices, even using the same velocity value, can vary widely. Real classification of perceived loudness would require signal processing of an acoustic signal, which I deliberately avoid in this thesis. I am forced instead to rely on the MIDI velocity measurement, which records the pressure with which a key was struck in the case of keyboards, and some approximation of force of attack when coming from a commercial pitch-to-MIDI converter following an acoustic source.

Accepting the MIDI velocity scale of 0 to 127, it was found that focus and decay had little effect on establishing any particularly more appropriate scale of values for measuring loudness. In fact, even with the MIDI scale, there seems to be no way to establish good thresholds for distinguishing velocity levels. One keyboard, played normally, may readily give a MIDI velocity of 100, for example, while another must be hammered with all the strength at one's command to get the same reading. There is no clear way to compensate for this algorithmically – consistently low readings, for example, could come from an exceptionally delicate piece. Placing the loudness threshold in the middle of the values actually seen, then, would simply classify half of the events as loud, when all of them are experienced as softly played.

For this reason, the dynamic agent is the only one which must be hand tuned for different instruments: an actual test with the physical instrument to be used must be performed, to find out what MIDI velocity readings will be registered for various types of playing. With that information at hand, a threshold corresponding to the instrument can be established, distinguishing soft from loud playing. Once a threshold has been chosen, the agent simply compares the MIDI velocity of incoming events to it, and reports a classification of loud or soft according to whether the event is above or below the threshold. Multiple-note events are classified as loud when more than half of the member notes are above the threshold, and as soft otherwise.

3.1.5 Density

The *Density* agent tracks vertical density. Vertical density refers to the number and spacing of events played simultaneously; this is in opposition to horizontal density, the number and spacing of events played in succession.

Horizontal density is treated by the *Speed* agent. The techniques of focus and decay are laid aside when considering vertical density; this is because the perceptual difference between linear and chordal textures seems to remain constant, regardless of the context. Similarly, though context does condition the perception of extension or clustering in chord voicings, the effect is still relatively weak. Because both of these aspects of vertical density (number of notes, spacing of notes) are represented with such low precision here, the coarser, more or less constant perception are being modeled. Consequently, the thresholds used to decide classifications of density and spacing are constant as well.

The classification performed by this agent can be considered as representing two related types of information: first, an event is judged to be a single note, or a chord. Second, if the event is a chord, the distance between extremes of the chord is considered, giving a classification of the chord's spacing. The first judgement, deciding if an event is a single note or a chord, is harder than it sounds. The main problem has to do with the way MIDI information is transmitted: serially, with notes from any simultaneity sent down the wire one at a time. Another problem is finding the boundary that separates fast trills from chords. At what point do the onsets of neighboring notes follow each other so quickly that they begin to be heard as a simultaneity? The answer to the second question, in *Cypher*, is 100 milliseconds. (Note that this constant was arrived at simply through experimentation with MIDI gear and does not represent any psychologically justifiable mark). Notes arriving within 100 msec of the first note in an event are considered part of the same event. However, establishing that threshold leads directly to the first problem: must we always wait 100 msec to find out if any more notes will arrive?

It took a surprisingly long time to find a good solution to this problem. The final implementation works as follows: The computer running *Cypher* is attached to an interface which receives and sends MIDI messages. Incoming MIDI messages are parsed, timestamped, and buffered in a MIDI driver which responds to the interrupts generated when a new message arrives. *Cypher* reads the buffered events from the MIDI driver in its main loop. First, a single MIDI event is read from the driver, and its timestamp recorded. This gives us

the start time of the current event. A counter, which will be incremented each time a null event[4] is read from the driver, is set to zero.

Now, MIDI events are repeatedly read from the driver and added to the *Cypher* event under construction as long as the following two conditions are true: the duration from the start time to the time of the most recent MIDI event read is less than 110 milliseconds, and the number of null MIDI events read is less than 50. These conditions are a consequence of the normal pattern of input accompanying a performed chord: timestamps for the notes in the chord are close to one another, but almost never simultaneous; however, a range of 110 milliseconds will almost always catch onsets which belong together.

Second, even MIDI packets with the same timestamp will sometimes be separated by null events. This is why one cannot simply keep reading events until the first null event. Allowing 50 null events to be read before giving up on additional data seems to capture almost all chords as chords, without introducing delays to system response while the program is waiting for additional data to straggle in. There is still some confusion in vertical vs. horizontal density: even with this method, fast trills will sometimes be read as groups of very compact chords. An additional rule could be added which would insert a chord boundary any time the same pitch was about to be added a second time to the same chord.

Once the initial event-gathering procedure has completed, the first step in the classification of density is indeed trivial: if an event's chordsize field is greater than 1, it is a chord, and a single note otherwise. Next, if it has been established that the density is greater than one, the agent classifies the spacing of the chord. To do this, the extreme notes are identified – then, the distance between extremes is measured. Chords falling within an octave are *octave1;* chords covering between one and two octaves are *octave2;* and chords spanning more than two octaves are *wide.*

---

[4] A *null event* is sent from the driver when an attempt is made to read it before a new MIDI event is complete in the buffer. Many null events can separate two MIDI events, even if the duration separating them is very small.

3.1.6 Attack Speed

The *Speed* agent classifies the temporal offset between the event being analyzed and the event previous to it in time. The offset is the duration in centiseconds between the attack of the previous event, and the attack of the analyzed event: this duration is sometimes referred to as the inter-onset-interval (IOI). Measuring the inter-onset-interval indicates the horizontal density of events; a low IOI separates events closely spaced in time.

The speed agent currently uses an absolute scale to classify the IOI, into one of four categories: events with an IOI longer than 2100 msec are classified as slow; those between 2100 and 1000 msec are classified as medium slow; those between 1000 and 300 msec are medium fast; and those shorter than 300 msec are fast. Note that the ranges decrease in size as the speed increases: the range of offsets for medium slow events is 1100 msec, down to a range for medium fast events which covers 700 msec.

In *Cypher*, the threshold at which horizontal density becomes vertical density is 100 msec: all raw MIDI events arriving within 100 msec of the onset of a *Cypher* event are grouped together in that *Cypher* event as a chord. In other words, *Cypher* events being constructed from MIDI data arriving from the outside world will never have an IOI less than 100 msec (*Cypher* events being wholly generated in the composition section, however, might be spaced more closely). The 100 msec threshold means that the range for fast offsets is limited to 200 msec (the difference between 300 msec, the upper bound, and 100 msec, the limit below which no offsets will be recorded).

Speed is a feature which clearly could benefit from a focus scale. Particularly with more precision, a sliding series of thresholds, able to reflect the relative speed variations according to context, would give a more lifelike representation of the experience of attack speed. Indeed, with adjustments by focus and decay, speed can be represented with somewhat less precision than would be needed without them, since the reduced classification set will always be referring to meaningful gradations. In other words, rather than having a large set of possible speeds, many of which remain unused in some

context, a smaller set, adjusting itself to the data arriving, could give a more accurate picture of what is actually happening.

3.1.7 Duration

The *Duration* agent classifies the length of *Cypher* events. A duration is the span of time between the onset, or Note On message, at the beginning of an event, and the Note Off message terminating it. Unlike the agents reviewed so far, the duration agent provides a classification to the featurespace characterizing the event *previous* to the one whose attack has just been recorded. This is because the level 1 featurespace analysis is done as quickly as possible after the initial attack of an event. A good part of the perceptual information accompanying a musical event is generated at the onset, particularly if that event has been represented in MIDI. If it is MIDI, and continuous controller information is ignored, *all* of the relevant information is present at the onset – except for the event's duration.

There are two options which could be taken in response to this problem: either analysis can be delayed until an event has finished, or the current event can be analyzed for everything but duration. The second solution has been chosen, since durations tend to remain relatively constant from event to event, and because the responsiveness of the system improves markedly if the current event is the one analyzed, and complemented by the composition section.

At present, an absolute scale is used to classify duration. Events whose duration exceeds 300 msec are classified as long, and the rest as short. The duration classification should be given more precision, and made available as a value relative to the current beat duration. Much as combining chord and key classifications yields the function of the chord in the key, combining durations and beat periods will yield an expression of an incoming duration in terms of whole and fractional parts of beats. Having durations available in such terms (one quarter-note, dotted eighth note, etc.) will produce sequences amenable to treatment by the pattern processes described in Chapter 6.

The implementation of focus on duration is problematic: a combination of sliding and fixed scales seems to be the correct way to proceed. For example, a grace note (a note of very short duration preceding some primary pitch), always has about the same length, and is always experienced as very short, no matter how long the surrounding durations are. So grace notes should always be classified as (very) short – the classification should not change as the primary pitches become shorter or longer. However, more typical note durations are sensitive to context: an eighth note surrounded by thirty-second notes will sound longer than an eighth note of the same duration surrounded by whole notes. Perhaps relatively extreme durations should receive constant classifications, and those in the usual duration range be subject to a more context-sensitive focal scale.

3.2 Harmonic Analysis

Categorical perception is a term used to describe the human propensity for separating some percepts into different distinct classes. The characteristic trait of categorical perception is that the perceiver will tend to classify a stimulus as belonging to one category, and then switch to another category, as the stimulus is continually varied along some dimension which defines class membership. In other words, the perceiver will always classify the stimulus as being one thing or the other, and never as an object with some qualities of both.

There are many examples of this kind of behavior in music listening. One example is the categorization of frequency information as belonging to certain discrete members of a scale, even if the physical vibrations of the frequency in question do not correspond directly to any particular member of that scale. Another example is rhythm perception, where the listener will tend to categorize attacks within an established rhythm as being one of a small number of simply related durations [Clarke87].

In the following two sections, I describe listening tasks resembling categorical perception: the first, harmonic analysis, maintains a theory of the current root and mode of the chordal area through which the music is passing, and, on a higher level, of the key to which those chords are related. The second task,

beat tracking, performs the categorical analysis illustrated in the example above: separating structural beat durations from the expressive and otherwise irregular spacings of event offsets.

3.2.1 Chord Identification

The goal of chord identification is to be able to determine, in real time, the central pitch of a local harmonic area. The harmonic sense implemented models a rather simple version of Western tonality. The choice of this particular orientation was made for two reasons: first, a pragmatic desire to test the function of categoric harmonic perception on easily understood tonal examples; and second, because a rudimentary understanding of tonality seems a reasonable capacity to give a program that attempts to deal with a wide variety of musical styles. To be sure, the tonal sense discussed here will be inadequate to describe many harmonic systems – a listener of many twentieth-century examples, in particular, should be programmed with a more wide-ranging vocabulary. However, the chord and key identification techniques described achieve a good measure of success within the target style, and mark out a path for adding supplemental harmonic competences.

With the aforementioned motivations, a series of tests were performed to determine a reliable method for finding the root and mode, in a simple tonic sense, of musical passages played in real time on a MIDI keyboard. The first method closely followed the model described in [Scarborough et al.89]. In this approach, connectionist principles are applied to the problem of analyzing harmonies. A neural network with twelve input nodes is used, where each input node corresponds to one of the twelve pitch classes in the tempered scale, regardless of octave. As each incoming MIDI note arrives, positive increments are added to theories associated with six chords of which the note could be a member, and negative increments are added to all other theories.

In Figure 13, we see an arriving 'c' MIDI note, shown as the leftmost input node at the bottom of the figure. Positive activation is seen spreading out from that node to the six simple triads (shown as output nodes at the top of the figure) of which the note could be a member: C major, C minor, F major,

F minor, Ab major, and A minor. The negative increments, sent on to all the other output nodes, are not shown.

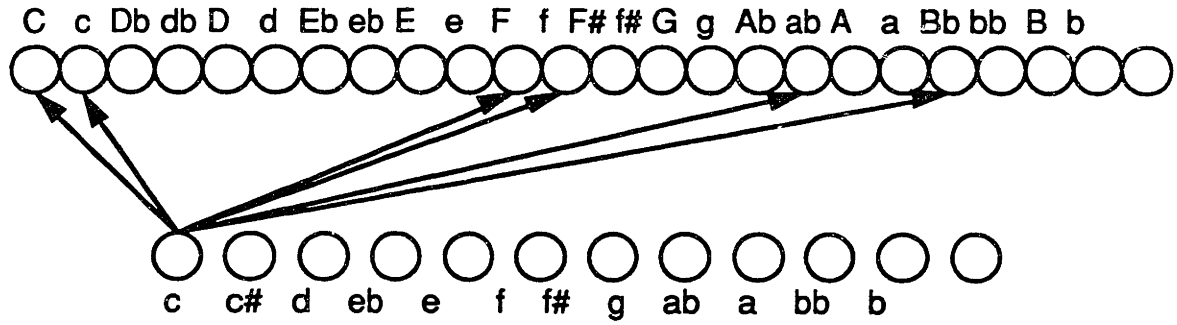


Fig. 13

The chord theories correspond to major and minor triads built on each of the twelve scale degrees. This is done not to restrict the harmonic discourse to major and minor triads, but to direct the analyzer to find a likely root and mode (major or minor) for any arbitrary chord. Therefore, the network will report a root of F not only for F major or minor triads, but for any chord which is more like F than anything else. Further, the output will be conditioned by the context, since a large amount of activation built up in some node will tend to keep that node dominant if subsequent input is ambiguous.

For each incoming note (octave is disregarded), the chord theories receiving positive increments are those for which the note could be the root (2 chords), the third (2 chords), or the fifth (2 chords). Roots are given the largest increment (5), fifths less (3), and thirds least of all (1). Major and minor chords of which the note is not a part receive negative increments. The magnitude of these vary according to the harmonic distance of the note from the chord.

In this test, the increments sent to each output node were developed by hand, through trial and error and a rudimentary consideration of traditional music theory. It would change nothing in the overall structure of the program to train a "real" neural net and use the acquired weights. In fact, it would be interesting to see how learned weights would compare with the ones I developed by hand. As we shall see, however, an important result for this

thesis lies in the way the behavior of a connectionist core can be improved through communication with several concurrent agents. The other significant difference to note between this method and normal neural net design is that all the weights involved are integers, and all node values are adjusted through addition: the motivation being a significant increase in speed. Again, a trained net could be converted to integer operation once the weights were acquired.

Relation of Pitch to Triad		Weight
Root	(C Major)	+ 5
Root	(C Minor)	+ 5
Maj 7	(Db Major)	- 15
Maj 7	(Db Minor)	- 15
Min 7	(D Major)	- 8
Min 7	(D Minor)	- 8
Maj 6	(Eb Major)	- 3
Maj 6	(Eb Minor)	- 3
Min 6	(E Major)	- 13
Min 6	(E Minor)	- 13
Perf 5	(F Major)	+ 3
Perf 5	(F Minor)	+ 3
Tritone	(F# Major)	- 16
Tritone	(F# Minor)	- 16
Perf 4	(G Major)	- 5
Perf 4	(G Minor)	- 5
Maj 3	(Ab Major)	+ 1
Maj 3	(Ab Minor)	- 15
Min 3	(A Major)	- 15
Min 3	(A Minor)	+ 1
Maj 2	(Bb Major)	- 6
Maj 2	(Bb Minor)	- 6
Min 2	(B Major)	- 15
Min 2	(B Minor)	- 15

Fig. 14

Figure 14 shows the weights sent to each of the 24 triads, in relation to an arriving 'c' (other pitches would send out the same weights to triads having the same relationship to them as these have to the 'c'). In developing these weights, it was found that the negative increments are in a sense more important than the positive ones: it is critical that a theory be cancelled out quickly when the harmony is moving from one chord to another. This is the reason that pitches with distant relationships to a given harmony, such as the

tritone or minor second, are given strong negative weights. Such cues will serve to move the agent off an established theory, when competing influences indicate that a harmonic motion is underway. Further, chord tallies are bounded at zero on the low end and 32 on the high end of the scale.

At any point in time, the root and mode of the current chord are taken to be those corresponding to the theory with the largest score. This method is a plausible first approximation of a tonic harmony analyzer; the unadorned version attains accuracy rates of better than 50% on Bach chorale examples. Successive refinements to this analysis method, however, were made not by concentrating on the weights associated with particular nodes, but rather by consulting a continually growing network of related features, in an effort to make the job of the chord analyzer simpler.

3.2.2 Connecting Additional Agents

Translating performed musical gestures to a MIDI stream causes simultaneous events (chords) to be serialized, sent down the wire one note after another. The effect of this serialization on the first method can be seen from the two trials shown in Figures 16 and 17. The data analyzed is taken from the first phrase of the chorale *Ach Gott, vom Himmel sieh' darein* by J.S. Bach (Fig. 15).

The graphs of data shown for each trial should be read as follows: the row of pitch names at the top shows the 24 possible chord theories, where an upper-case letter indicates the major mode, and lower-case indicates minor. So, the first two columns correspond to C major and C minor. The leftmost column shows incoming notes; these are considered without octave references, and so are shown by pitch name only.

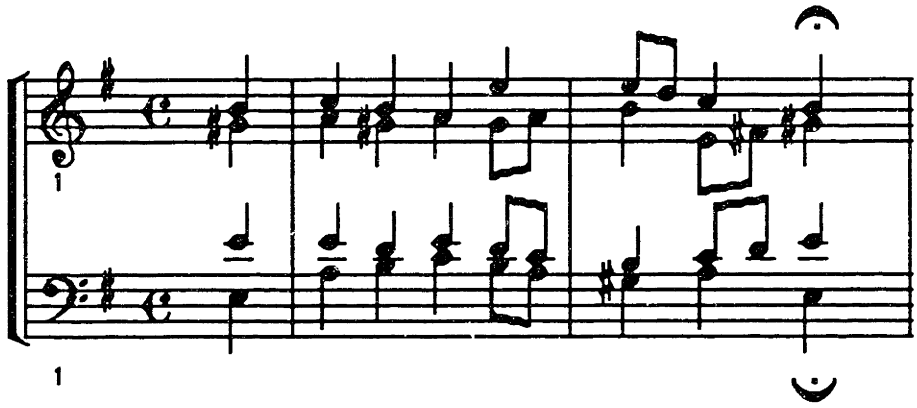


Fig. 15

Reading across each row from the incoming pitch name, the tally associated with each chord theory is shown. Those theories with the highest value are followed by the sign ']' – e.g. a value of `5]` for a given theory means that that theory's tally equals 5, and that 5 is the highest tally of all theories in that row (though there may be others with the same score). Finally, the rightmost column shows the *confidence* of the analyzer in the winning theory (or theories). Confidence is simply a measure of the strength of the winning theory relative to all the scores in that row, as in: `certain = (high_theory/total_points_in_row)*100`. Therefore, the more a winning theory captures the points available at any one time, the higher its confidence rating will be.

In the first trial, the E major chord at the beginning of the chorale was arpeggiated with the notes played from bottom to top (to force an evaluation in that order); in the second trial, the same chord was arpeggiated from top to bottom. The absolute value for a theory of E major, after all four notes have passed, is the same in both trials (14). The certainty rating is slightly different (41 in the first trial, 32 in the second). However, the intermediate results are strikingly different. The second trial begins with a theory of B (major or minor), and passes through g sharp minor before settling on E major. The first trial produces E major (or, at the outset, minor) throughout.

	C	c	C#	c#	D	d	Eb	eb	E	e	F	f	F#	f#	G	g	Ab	ab	A	a	Bb	bb	B	b	
e	1			1					5]	5]									3	3					27
e	2			2					10]	10]									6	6					27
g#			3	5					11]			1					5	5							36
b									14]	3					1			6					5	5	41

• played bottom to top

Fig. 16

	C	c	C#	c#	D	d	Eb	eb	E	e	F	f	F#	f#	G	g	Ab	ab	A	a	Bb	bb	B	b	
b									3	3					1			1					5]	5]	27
g#			3	3					4			1					5	6]					2	2	23
e	1			4					9]	5									3	3					36
e	2			5					14]	10									6	6					32

• played top to bottom

Fig. 17

The same phenomenon will be observed for any chord when the order of evaluation is changed. Each successive pitch serves to further direct a search through the space of chord theories; different orderings of pitches necessarily result in different search paths. We are able to skirt the problem of internal path deviations, arising from evaluation order differences, by consulting the density agent. When that communication is added, the listener knows which notes are part of a chord, and which have been played separately. Therefore, a simple refinement of the chord agent allows it to reserve classification until all the notes of a chord have been processed. The internal path through the chord theories is rendered irrelevant, since it is never seen by the rest of the system. Only the final theory, which remains reasonably stable (though certainty ratings are seen to change from one ordering to another), is broadcast.

In the next improvement, a communication path was established to the register agent. In Western tonal music, significant harmonic information tends to be presented in relatively lower registral placements. The bass voice

of a four-part texture is more likely to be consonant with the prevailing harmony than the soprano: higher voices often include pitches dissonant to the harmony as passing tones, or in ornamentation. Therefore, the chord identifier was modified to give greater weight to information coming from the lowest register than to messages from higher registral areas.

We can see this illustrated in the data of Figure 18: the pitches which have been found in the lowest register are marked by an asterisk in the far right column. The effect of these pitches on the chord theories can be seen from their greater weight; the bass 'e' from the first chord, for instance, gives 1.4 times more emphasis to the E major theory than does the 'e' higher up.

```
    C  c  C# c# D  d  Eb eb E  e  F  f  F# f# G  g  Ab ab A  a  Bb bb B  b
g#        3  3           1        1           5] 5]                    27
e   1        4           6] 5                    3  3                 27
b                        9] 8              1     1              5  5  31
e   2        2           19]18                   6  6                 35 *
___
a              6  6       9  8  2        2        16]16]               24 *
e   1        1           14 13                   19]19]               28
c   6  5                  1     3  3              1   4 20]            46
```

• bass emphasis

Fig. 18

Finally, the beat agency is consulted. The heuristic here is that pitch information on the beat is more likely to be consonant with the dominant harmonic area than pitches off the beat. As we shall see in more detail in Chapter 5, the beat agency conducts an initial pass over the data before the chord agency is called. So, the chord process is able to obtain good information from the beat tracker about the current event. Weights associated with events on the beat are given 1.1 times the emphasis of events off the beat.

The chord identifier is able to return two kinds of information in response to query messages: the first is the current theory number. The theory number conveys both the root and mode of the chord, since twenty-four separate theories are maintained. Major and minor mode versions of the same root are paired, such that C major is theory 0, C minor is theory 1, and so on.

Because of this ordering, querying processes can find the mode of the chord by taking the theory number modulo 2 (even theories are major, odd ones are minor), or find the root of the chord from dividing the theory number by two. The second piece of information returned is the confidence rating of the analyzer in the theory. This can be used by other agents to discard theories with low confidence levels.

3.2.3 Key Identification

In Western tonal harmony, there is a clear hierarchic relationship between the concepts of chord and key. Chords are local harmonic areas, dominated by the root of a collection of pitches in a small temporal area. In fact, the term *chord* is used here as a convenience; the important distinction is that we are discussing the harmonic root of a local level – all of the discussion following about chords could refer equally well to contexts of lower vertical density. For instance, there is a level of harmonic activity present in a flute solo, necessarily presented on a one-note-at-a-time basis, which is identical to that which I will here refer to as the chord level. On a higher level, chords themselves are related to keys – harmonic complexes whose central pitch affects the perception of tonal relations through longer spans. In *Cypher*, these scales of harmonic influence are reflected in the separation of chord and key analyses, and the performance of these tasks on different listening levels. As we have seen, the local harmonic analysis (chord identification) is performed by an agency invoked on level 1; a second agency, concerned with longer term pitch centers (key identification), resides on level 2.

The level 2 harmonic analysis (hereafter referred to as key identification) has, like the chord agency, a connectionist core. The input nodes of the chord net are activated by the pitch classes of all *Cypher* events present in the listener's input stream. The key identification network has twenty-four input nodes, which are activated by the chord classifications from the harmonic analysis agency one level down. Each arriving chord classification, one per event, will activate an input node, which in turn spreads activation among the twenty-four output nodes, interpreted as corresponding to the minor and major modes built on each of the twelve scale degrees.

The key theories which are most positively influenced by an incoming major chord are those for which the chord could be the tonic, dominant, or subdominant. Other increments vary by the mode and scale degree of the chord, and the mode of the prevailing key theory (since chords on some scale degrees will be minor in the major key, and major in the minor key). In Figure 19, we see twelve input nodes represented, for major mode chord reports from the twelve scale degrees. The complete network has twenty-four input nodes. In the figure, positive activation is spreading from the input to key theories for which the chord is tonic, subdominant, or dominant. Negative activation, which is simultaneously spreading from the same input node to all other key theories, is not shown.

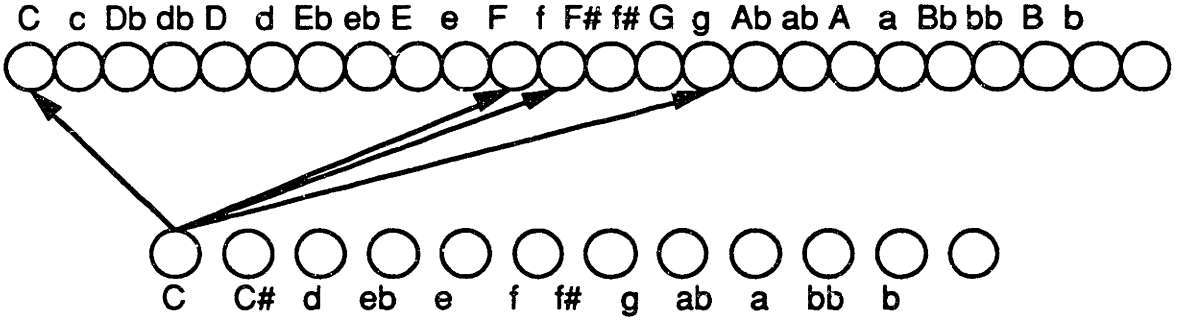


Fig. 19

The weights tabulated below were arrived at through trial and error, rather than through a machine learning technique such as the back propagation algorithm. Again, these weights could easily be replaced by a new set developed through back propagation training. In developing them by hand, it was noticed that, like the chord weights, finding good negative increments is in a sense more important than establishing positive ones. In this scheme, only four theories receive positive increments from a chord input. The other twenty theories receive negative or null increments. Well chosen negative weights are important because they are the ones which will break down an established theory and make room for a new one to emerge. Particularly when tracking keys, one theory will tend to remain strongest for a long period, but must be supplanted by another rather quickly. Negative weights, coupled with an upper bound on theory strength, keep any theory from

becoming so dominant that the analysis will become sluggish at finding changes of key.

Following are the weights sent to each of the 24 key theories, in relation to an arriving **C major** chord (other chords would send out the same weights to keys having the same relationship to them as these have to **C major**) :

Relation of Chord to Key		**Weight**
Root	(C Major)	+ 3
Root	(C Minor)	- 3
VII	(Db Major)	- 5
VII	(Db Minor)	- 5
bVII	(D Major)	- 6
bVII	(D Minor)	- 7
VI	(Eb Major)	- 6
VI	(Eb Minor)	- 7
bVI	(E Major)	- 5
bVI	(E Minor)	- 7
V	(F Major)	+ 2
V	(F Minor)	+ 2
bV	(F# Major)	- 7
bV	(F# Minor)	- 7
IV	(G Major)	+ 1
IV	(G Minor)	0
III	(Ab Major)	- 5
III	(Ab Minor)	- 7
bIII	(A Major)	- 7
bIII	(A Minor)	- 4
II	(Bb Major)	0
II	(Bb Minor)	- 1
bII	(B Major)	- 7
bII	(B Minor)	- 7

Fig. 20

As in the chord network reviewed earlier, key weights are integers, and affect the nodes to which they are connected by addition. Further, key theory tallies are bounded on the low end at zero, and on the high end at 60, to keep any theories from becoming so dominant, or so weakened, that they will be kept from competing when a change of context occurs.

Following are the weights sent to each of the 24 key theories, in relation to an arriving **C minor** chord (other chords would send out the same weights to keys having the same relationship to them as these have to **C minor**) :

Relation of Chord to Key		**Weight**
Root	(C Major)	- 4
Root	(C Minor)	+ 3
vii	(Db Major)	- 6
vii	(Db Minor)	- 6
bvii	(D Major)	- 7
bvii	(D Minor)	- 7
vi	(Eb Major)	0
vi	(Eb Minor)	- 6
bvi	(E Major)	- 7
bvi	(E Minor)	- 6
v	(F Major)	- 1
v	(F Minor)	0
bv	(F# Major)	- 7
bv	(F# Minor)	- 7
iv	(G Major)	- 6
iv	(G Minor)	0
iii	(Ab Major)	+ 1
iii	(Ab Minor)	- 7
biii	(A Major)	- 7
biii	(A Minor)	- 7
ii	(Bb Major)	0
ii	(Bb Minor)	+ 1
bii	(B Major)	- 7
bii	(B Minor)	- 7

Fig. 21

3.2.4 Key Agency Connections

The connectionist part of the key identification process just reviewed is again at the core of a larger agency, in which other featural and higher-level agents are linked together with the key net to form a complete, more accurate reading of the current harmonic context. Before describing the agents making up the key agency, let us be precise about the level of harmonic activity we are trying to describe. We have noted that the "chord" agency refers to local harmonic areas, whether the pitches making up those areas are actually played simultaneously or not. Similarly, the agency we are now describing tries to find a root pitch, and scale mode, for harmonic activity at the phrase

level. We can imagine higher level harmonic analyses, which would take the results of the "key" agency and, for example, look for the dominant pitch and mode over entire movements of a composition.

The first agent added to the key net is the one tracking vertical density. There is information arriving from the chord agency with every input event. However, it is of interest to the key analyzer to know the vertical density associated with arriving chord reports. Many Western musical styles will present non-harmonic pitches in passing tones, ornaments, or other kinds of linear embellishments to a more chordal texture. This heuristic is represented in the key agent by giving greater weight to harmonic information presented in simultaneities. Linear pitch material will still contribute to the analysis, but will accumulate more slowly than chordal input. This communication with an analysis of vertical density corresponds well with the kind of behavior we would expect from the three possible textural types: 1) all chords: harmonic analyses carry the same weight, since there is no difference in vertical density; 2) completely linear: harmonic analyses are arrived at, but more slowly than in denser textures; 3) combinations of linear and chordal material: denser material will advance harmonic analyses more quickly than does the (typically) more non-harmonic linear music.

Information flows from the chord identification to the key analyzer; an improvement to the chord identification algorithm was made by including a feedback path from the key to the chords. The chord finder often produces ties between two rival chord theories. One typical case is confusion between major and minor theories based on the same root. This situation can not be disambiguated until a third comes along. However, ties between other theories frequently arise, particularly when the analysis is in transition from one dominant theory to another. In these instances, the current key analysis is consulted. A table of probabilities for each chord in each key is consulted, and the most likely chord for the current key is chosen as the output of the chord identification process.

Chord identification in *Cypher* is performed by an *agency* of small, relatively simple processes. This agency includes the main chord analysis method, and the register, density, and key analysis agents. Each of these agents performs its

own, specialized task, and communicates the results of this analysis to other agencies which profit from a broader context. The chord agency itself could benefit from an expanded repertoire of contributing agents: duration and dynamic could also help indicate relative importance among incoming events, such that the pitches associated with those events receive greater weight in the chord identification process.

In Figure 22, we see the connections between the network core of the chord agency, and other contributing agents. Dynamic and duration are connected by dotted lines to indicate their potential, but not actual, contribution. Beat tracking, covered in the next section, has a two-way connection to the chord net. The two-way link indicates that both agencies are consulting each other. Chords and keys are similarly connected in both directions. More about two-way links will said in Chapter 5. Finally, the density and register agents are shown sending information to the chord and/or key analyses.

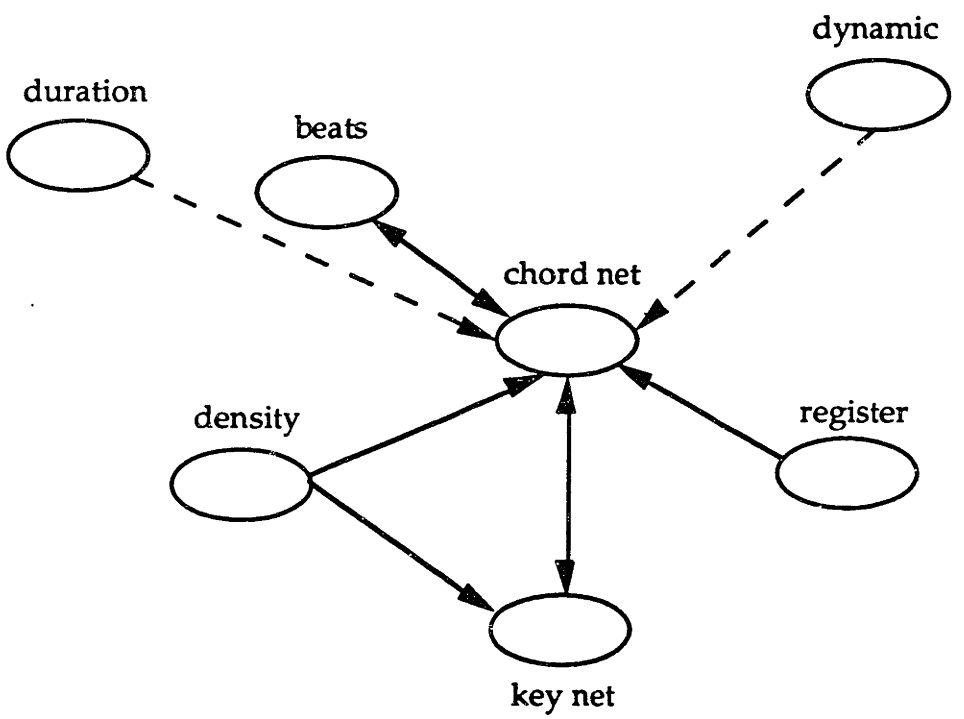


Fig. 22

3.3 Beat Tracking

Beat tracking is the process of finding a duration to represent the perceived interval of a beat, described as that level of temporal periodicity in music to which a listener would tap a foot, or a conductor move a baton [Chung89]. The beat tracking implemented here assumes no prior knowledge of the input; in other words, no pattern matching can be done between the input and some stored representation of it. In its simplest form, beat tracking is only concerned with finding the beat interval. The closely allied perception of meter, in which some beats are experienced as "stronger" than others, is left for future research.

3.3.1 Multiple Theory Technique

Cypher accomplishes beat tracking with an agency built around a connectionist core tuned to find beat periodicities in MIDI data. The beat agency is quite similar to the agencies for chord and key identification in this respect, and in fact consults many of the same agents that contribute to the other tasks. Further, the beat agency communicates with the chord, key, and phrase agencies the progress of its analysis, and in turn consults the feature agents and the chord agency to assist in determining a plausible beat interpretation of the incoming MIDI stream.

The connectionist core of the beat agency maintains a large number of theories, which represent possible beat interpretations. Theories of beat periods are maintained for all possibilities between two extremes judged to be the limit of a normal beat duration. The limits in this case were taken from the indications on a musical metronome, which has historically been found sufficient for typical beat durations. These limits are: 40 beats per minute on the slow end, and 208 beats per minute on the fast end. Measured in milliseconds, metronomic beat offsets then fall in the range of 288 to 1500 msec. In the multiple theory algorithm, separate theories are maintained for all possible centisecond offsets within this range; in other words, offsets from 28 to 150 centiseconds (a total of 123 possibilities) are regarded as possible beat periods.

There are two parts to a beat theory : the theory's points, and an expected arrival time. There are also two ways for a theory to accrue points : first, each incoming event spawns a list of candidate beat interpretations. The members of this list are all awarded some number of points. Second, the arrival time of incoming events are checked against the predictions of all non-zero theories. Theories whose prediction coincides with the real event are given additional points. Figure 23 illustrates this principle: the top row of arrows represent actual incoming MIDI events. The lower row of arrows represent the event arrivals predicted by a beat theory. When the arrows of actual events line up with the predicted arrivals, the corresponding theory is positively incremented. When no actual arrow aligns with a beat prediction, the theory loses strength.

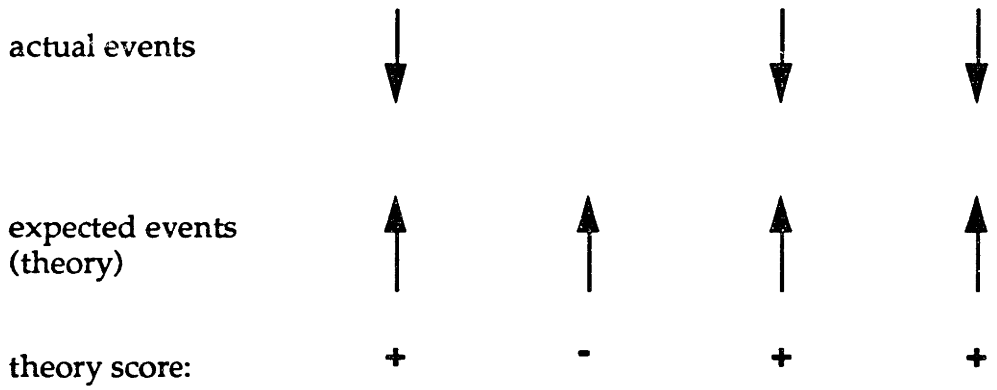


Fig. 23

The algorithm works as follows: when an event arrives for analysis on level 1, it is passed to the beat tracking agency. The first thing the tracker does is to examine the expected event arrival times of all theories. If the real arrival coincides with an expected arrival for any non-zero theory (within a small margin to allow for performance deviations), points are added to that theory's score. If the real arrival comes before an expected arrival, the real offset is subtracted from the expected offset, so that on the next event the agent will still be looking for a "hit" at the same absolute time. If the real offset arrives later than an expected offset, points are subtracted from that theory's score.

The heuristic here is that syncopations are unlikely: that is, true beat pulses will usually have events aligned with them.

The beat tracker can return two different values in response to queries: 1) the absolute time of the next expected beat, and 2) the current beat period. The first response allows other processes to schedule events to coincide with the expected next beat. The second value can be used to schedule events at regular intervals corresponding to the calculated beat period.

3.3.2 Generating Candidate Interpretations

The first part of the beat tracking algorithm uses the syncopation heuristic; theories whose estimated time of arrival coincides with a real event are rewarded, and those theories with ETAs *before* the incoming event (i.e., the incoming event is syncopated with respect to the theory) are penalized.

The second step in the algorithm looks for the five most likely beat interpretations of the incoming event offset. Two candidates are found from the offset itself, and from the offset of the previous event. Then, a set of factors is used to generate the rest: the members of the set are each multiplied in turn with the offset of the incoming event, to produce a possible beat duration. If the resulting duration is inside the acceptable range (i.e., on the metronome), it is added to the list of candidates. If the factored offset is off the metronome (outside the approved range), it is rejected, and the next factor on the list tried. This process continues until five candidate interpretations are found.

Initially, two candidates are generated independently of the factors set. The first interpretation evaluated is always the offset associated with the incoming event; if that offset by itself is within the accepted range, it is given first place on the list. The second interpretation evaluated is the result of adding together the current and previous event offsets. The idea here is that the true beat duration will often not be directly present in the input, for example in the case of a regularly recurring quarter/eighth duration pattern (such as might arise in 6/8 time). In such a situation, the true beat duration is a dotted quarter; that duration, however, will be rarely present in the input. It would

arise from the factorization process, as we shall see in a moment. However, it is quite effective to consider the simple possibility of adding adjacent offsets, since (as in the example) these will often yield the appropriate duration. For the second candidate, then, the current and previous offsets are summed, and added to the list if the result is on the metronome.

All other candidates are generated from the factors set. This set assumes the usual Western rhythmic subdivisions of two or three, or multiples of two or three, to a beat. The factors include such interpretations as that the incoming offset represents half the beat duration; twice the beat duration; 1/3 the beat duration; 1/4 the beat duration; etc. Following is the factors set :

```
static float factors[FACTORS] =
{ 2.00, 0.50, 3.00, 1.50, 0.66, 4.00, 6.00, 1.33, 0.33, 0.75 };
```

Let us look at a simple example:

```
          0          1          2          3          4      R      E

38   38[  6]   53[  5] 114[  1]   84[  1] 150[  0]    345   380a
39   38[ 12]   80[  6] 115[  2]   53[  2] 150[  0]    382a  420b
37   37[ 18]  113[  7]   78[  7]   54[  3] 148[  1]    422b  459c
39   38[ 24]   77[ 12] 115[  8]  148[  2] 150[  0]    459c  496d
38   38[ 30]   77[ 13] 114[  9]   53[  5] 150[  0]    496d  534e
37   37[ 36]   76[ 18] 112[ 14]  148[  1] 150[  0]    532e  569f
35   36[ 42]   74[ 19] 108[ 15]  144[  2]   52[  1]    568f  604g
38   37[ 48]   73[ 24] 111[ 16]  144[  3] 150[  0]    605g  642
77   37[ 41]   75[ 30] 107[ 13]  150[  0] 149[  0]    683   720
78   76[ 40]   38[ 34] 112[ 14]  150[  0] 149[  0]    760   836
```

Fig. 24

The trace above was generated in real time from an analysis of a live performance. The rhythm played was the following, where 8 stands for a played eighth note, and 4 a quarter note, in moderate tempo: 88 88 88 88 4 4 4. There are one fewer rows in the activation trace than in the rhythm, because two notes must pass to give an initial inter-onset-interval.

This and subsequent beat theory activation traces are to be read as follows: the first column indicates the offset in centiseconds of the event being analyzed,

measured from the attack of the previous event. Then, the trace has five columns marked from 0 to 4. In these columns are recorded the five beat theories with the highest scores. For each theory, there are two values shown. The first is the projected beat duration for the theory; the second (following the duration in brackets) is the points currently associated with that theory. So, an entry of `38[ 6]` would indicate a beat duration theory of 38 centiseconds, which currently has a score of six.

In the above example, we see the eighth note duration hovering around 37 centiseconds, with a typical performance spread covering a total of about 4 centiseconds. The quarter notes also show relatively constant values, more or less twice that of the eighth notes. The eighth note tempo is initially accepted as the beat offset, because it is "on the metronome." However, when the quarter notes arrive, the quarter note duration quickly takes over as the beat duration. This process is seen clearly at the input of 77, the second quarter note (at the end of the first quarter note duration). Because this duration arrives later than the expectation for the eighth note pulse, the eighth note theory loses seven points (drops from a score of 48 down to 41). The input falls directly on the expected quarter note arrival time, however, and this theory gains six points. When the final quarter note comes (at the end of the second quarter note duration), the same thing happens, and the quarter note beat duration takes over first place.

The most interesting entries on the activation trace are those at the far right of the figure, the columns marked 'R' and 'E'. These mark the "real" and "expected" event arrival times. The "real" time is the clock time recorded when the incoming event was analyzed, and the "expected" time is the arrival time calculated for the next event after the analysis. Therefore, the expected time in any row should match a real time in some later row, if the tracker is working correctly. In the figure, I have marked by hand the correspondences between expected and real arrival times. Lower-case letters are placed beside correctly predicted event times, with the same letter next to the prediction and the actual event. We see a quite acceptable alignment of expected and real event arrivals, up to the moment when the performance changes from eighth notes to quarter notes. The events predicted to arrive at times 642 and 720 never do arrive: the real events are syncopated with respect

to them. By the end of the example, the quarter note pulse has taken over, and the next expected arrival is anticipated at one quarter note duration later.

3.3.3 Accommodating Performance Deviations

Even in this most simple example, which should present no performance difficulties whatever, we see the inevitable onset deviations which mark human playing. Such deviations will be even more pronounced with more difficult music, or when a performer is purposely adding expressive transformations to the offset times.

Because of these variations, before points are awarded, the algorithm searches the immediate vicinity of a candidate theory for other non-zero theories. Non-zero neighbors are taken to be a representation of the same beat, but with a performance deviation from the candidate. If such a neighbor is found, the tracker asserts that the composite theory is midway between the neighbor and the candidate, and adds together the points from the neighbor, and those due to the candidate. Then the neighbor and candidate theories are zeroed out, leaving one theory in that vicinity – the average duration calculated from previous and current inputs.

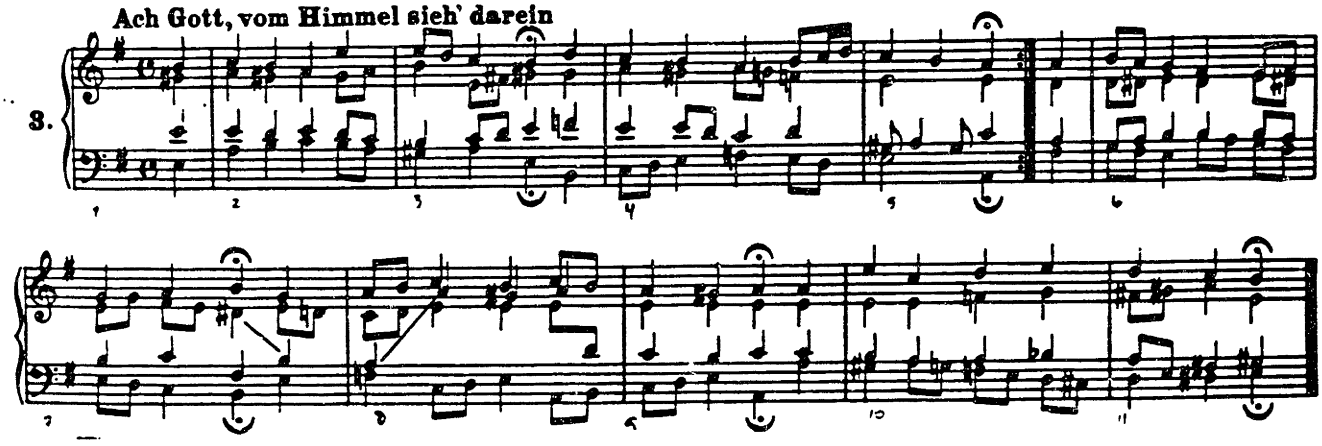


Fig. 25

We can clearly see this "zeroing in" behavior in the next trace, taken from a live performance of the Bach chorale example shown in Figure 25. The tracker maintains the quarter note pulse as the leading theory throughout. However, because of the performance deviations, the period of the beat duration moves slightly throughout the trace, starting at 74 centiseconds, and moving as low as 68, with an average value around 70. Again, I have marked correspondences between expected and actual arrival times with lower case letters – identical letters show successful predictions of event arrivals.

The tracker does a respectable job on this example: 13 of 14 quarter notes are correctly predicted, 12 of them within 4 centiseconds. It is interesting to note where the predictions break down: the sixteenth notes at the end of measure 4 are performed a little slowly, which leads to an incorrectly predicted downbeat of measure 5. By the second beat of measure 5, however, the prediction again matches reality.

	0	1	2	3	4	R	E
74	74[6]	148[1]	111[1]	94[1]	150[0]	382	453a
70	72[17]	146[6]	150[0]	149[0]	148[0]	449a	521b
71	71[28]	143[7]	106[1]	150[0]	149[0]	520b	591c
67	69[34]	140[12]	150[0]	149[0]	148[0]	589c	657
35	69[35]	140[13]	35[6]	102[1]	52[1]	624	658d
34	69[40]	138[14]	34[12]	102[2]	150[0]	658d	727
34	68[41]	34[18]	137[15]	102[7]	51[1]	695	729e
35	68[46]	34[24]	138[20]	103[8]	150[0]	730e	798
35	69[52]	34[30]	139[21]	104[9]	52[1]	765	799f
36	70[57]	35[36]	141[22]	106[14]	150[0]	803f	872g
73	71[68]	35[29]	143[27]	107[15]	150[0]	876g	947h
70	70[74]	143[28]	35[22]	106[8]	150[0]	947h	1016i
37	72[80]	145[29]	36[28]	106[9]	55[1]	984	1019
32	70[85]	136[34]	34[34]	101[10]	150[0]	1017i	1086j
38	70[91]	36[40]	136[31]	107[15]	53[5]	1056	1087
33	70[96]	34[46]	134[32]	103[16]	150[0]	1089j	1158
35	69[102]	34[52]	137[33]	104[17]	52[1]	1124	1157k
32	68[107]	33[58]	132[26]	100[22]	150[0]	1157k	1224
36	68[108]	34[64]	138[27]	104[23]	54[1]	1193	1225
18	70[109]	35[65]	138[24]	106[24]	54[6]	1210	1226
21	70[110]	37[70]	138[21]	106[21]	19[12]	1232	1302
34	69[111]	35[76]	104[26]	137[22]	56[12]	1267	13011
35	69[116]	35[82]	104[27]	138[15]	54[13]	13011	1370
36	70[117]	35[88]	106[28]	141[16]	54[6]	1337	1371m
38	72[122]	36[94]	110[33]	141[13]	52[11]	1377m	1449

Fig. 26

The Bach example is of limited rhythmic complexity, but demonstrates the kinds of rhythmic behavior the tracker can handle well. This method is more erratic with rhythmically complex input; particularly passages with fast figuration around a slow underlying beat tend to confuse it (such examples are an exaggerated form of the sixteenth note mistake noted above). I am confident that significant improvement to this tracker could be achieved with better weights, more sophisticated candidate selection, and added constraints indicating typical rhythmic progressions. Again, the point of this thesis was not to develop the perfect beat tracker; but rather, to include beat tracking in a network of many competences, all able to help each other achieve better performance on their specialty. The chord agency and dynamic agent are two entities whose reports affect the calculation of beat periods. In Chapter 5, I will explore in more detail the relationships with other agents into which the beat tracker joins.

3.4 Grouping

One of the most critical functions of the level 2 analysis is to find structural groups in streams of lower level *Cypher* events. The groups found on level 2 correspond to what are usually called *phrases* in music theoretic terms. Phrases are musical sequences, commonly from around 2 to 10 seconds in duration, which cohere due to related harmonic, rhythmic, and textural behaviors. The level 2 listener detects boundaries between phrases by looking for discontinuities in the output of the level 1 feature agents. Such discontinuities contribute, with different strengths, to the perception of phrase boundaries. In *Cypher*, the weighting of feature discontinuities as cues for phrase boundaries is as follows :

Feature	**Weight**
Density	1
Register	1
Speed	4
Dynamic	1
Duration	2

The phrase boundary agent collects information from all of the perceptual features, plus the chord, key, and beat agencies. When a discontinuity is noticed in the output of a feature agent, the weight shown in the table above for that feature is summed with whatever other discontinuities are present for the same event. When the sum of these weights surpasses a threshold, the phrase agent signals a group boundary. Note that this signal will correspond to the initial event of a new phrase; the discontinuities are not noticed until after the change.

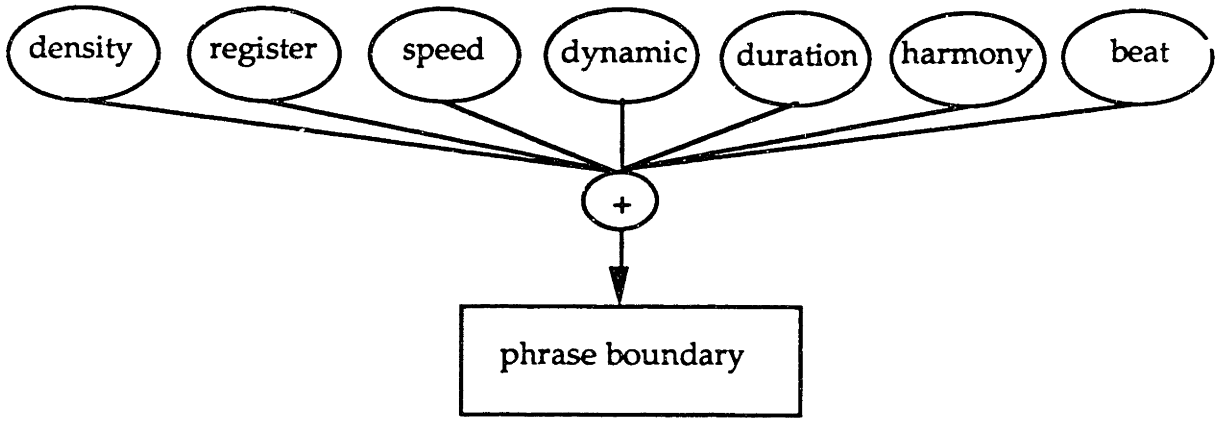


Fig. 27

The remaining feature dimension, harmony, is treated somewhat differently. The weight of the harmonic analysis is decided by the function of the current chord (local harmonic area) in relation to the current key. In other words, a chord analysis of F major will not be considered in isolation, but considered as a functional harmony in some key. If the current key were also F major, for instance, the chord would have a tonic function (or function number zero, in the *Cypher* numbering scheme). Following the conventions of Western tonal harmony, tonic and dominant functions are given more weight as potential phrase boundaries than are chords built on other scale degrees. The following table lists the phrase boundary weights given to the various chord functions:

Function	I	i	bII	bii	II	ii	bIII	biii
Weight	4	4	1	1	2	2	1	2
Function	III	iii	IV	iv	#IV	#iv	V	v
Weight	2	0	3	3	1	1	5	5
Function	bVI	bvi	VI	vi	bVII	bvii	VII	vii
Weight	1	2	2	0	1	2	2	2

Fig. 28

The phrase boundary analysis relies heavily on the progressive perspective; neighboring events in time are compared, and split into different groups according to their similarity or dissimilarity. Another manifestation of this reliance is an extension of the functional harmony contribution just described: adjacent events' functions are checked to see if they manifest a dominant/tonic relationship. That is, if the harmonic event for the last event has a function a perfect fifth above that of the current event, the evidence for a phrase boundary is strengthened. Another contribution comes from the beat tracker: events on the beat are given more weight as potential phrase boundaries than events not landing on a predicted beat point.

Further, the phrase boundary calculation implements a version of the techniques of focus and decay. Initially, the phrase boundary threshold is set to a constant. When a new phrase boundary is found, the number of events in the new phrase is checked: if there are under two events in a phrase, the threshold is incremented – phrase boundaries are being found too quickly. On the other side, there is an maximum limit to the number of events in a phrase. If the event count passes this limit, a phrase boundary is declared, and the threshold is set lower – phrase boundaries are not being found quickly enough. In this case, the heuristic for moving the threshold up seems to work well. The decay part of the rule seems too arbitrary; there should be a more musical way to decide that phrases are too long than by counting events.

3.4.1 An Extended Example

In this section, we will examine analysis traces from the key and phrase detection agencies generated from the processing of a real-time performance

of a Bach chorale example, shown in Figure 25. This chorale was chosen because it presents a challenging tonal language, stretching the capabilities of the harmonic analysis implemented. Simpler, more easily handled examples could have been presented instead, but in the following exposition, the ways in which the process falters are as instructive as the successes.

First, consider the output of the key agency. Incoming chord agency reports are shown in the leftmost column. The activation associated with each key theory is shown in the 24 following columns. The highest activation for a key theory in any row is followed by a ']' symbol. So, in the very first row, the chord agency reported an E major chord, spawning a key theory of E major. Each measure in the original score has been marked by hand in the trace with a dash in the chord column. Accordingly, there is a dash after the first E major report, since that measure contains only the pickup chord. The dashes are a convenience for comparing the trace to the score.

	C	c	C#	c#	D	d	Eb	eb	E	e	F	f	F#	f#	G	g	Ab	ab	A	a	Bb	bb	B	b
E									4]										2	2			1	
- m2																								
a											1				1	1				7]				
b															2				1	8]				5
a											1				3	1				13]				
b															4				1	14]				5
a											1				5	1				19]				
- m3																								
E									4										2	21]			1	
b									3						1				3	22]				4
a											1				2	1				27]				
D					4										4	3			1	27]				
E					4				4										3	29]			1	
b					4				3						1				4	30]				5
- m4																								
a					3						1				2	1				35]				
d	1	1				4					1				1	1				35]	1			
E						3			4										2	37]			1	
E						3			6										3	38]			1	
F	1										4									31]	2	2		
G	2	1									4				2					29]				
e	3				1	1				5					2					29]				
c		5															1			29]	1	1		
d	1	6				4														29]	2			

	C	c	C#	c#	D	d	Eb	eb	E	e	F	f	F#	f#	G	g	Ab	ab	A	a	Bb	bb	B	b
- m5																								
E						3			4										2	31]			1	
A					1	4			4										4	30]				
E					1	4			6										5	31]				
E					1	4			8										6	32]				
a						4			1		1				1	1			1	37]				
D					4	1									3	3			2	37]				
- m6																								
D					8										5	5			3	37]				
f#					9				1	1				5					3	30]				
e	1				10	1				6									2	30]				
B					4				2	8			1						2	29]			4	
A					5	1			2	8									4	28]			1	
e	1				6	2				13									3	28]				
F#			1		1					12			4							21]			2	2
- m7																								
e	1				2	1				17										21]				2
G	3	2			3	1				13					4					16]				
G	4	3			3	1				11					6					14]				
G	5	4			3	1				9					8					12]				
B									2	11			1		3					11]			4	
e	1				1	1				16]					3					11				
e	2				2	2				20]					3					11				
- m8																								
F	3									13]	4									4	2	2		
d	4	1				5]				4	4									4	3			
C	8]										6	2			1						3			
d	9]	1				4					6										4			
E	4]					3			4		1								2	2			1	
a	4					3					2				1	1				7]				
b															2				1	8]				5
- m9																								
C	4										2	2			3					4]				
C	6]										3	3			3					2				
E	1								4]										2	4			1	
a	1										1				1	1				9]				
a	1										2				2	2				14]				
- m10																								
E									4										2	16]			1	
a											1				1	1				21]				
C	2										2	1			1	1				19]				
d	3	1				5					2					1				19]	1			
e	4				1	6				4										19]				
g	3					6	1			4	1	1				5				10]				
C#			2			4				1			1	1		2				7]				
- m11																								
D					4	1									2	4			1	7]				
E					4				4										3	9]			1	
f#					5]				5	1				5					3	2				
E					5				9]										5	4			1	

Fig. 29

Notice the first key change, in measure 7. The tonic moves from a minor to e minor, which has taken over by the middle of measure 7. The move back to a

minor is more circuitous; the agency notes the ambiguity at the onset of measure 8, with the F major chord, and abandons e minor for C major. The E major to a minor transition between the second and third beats, however, is enough to re-establish a minor. Another instability occurs at the beginning of measure 9, when a first inversion a minor chord is misread as C major. These reports, and the B major report earlier, would be ignored, since key changes which are not confirmed by at least two successive reports are discarded. The tonal ambiguity of the end of the piece is instructive: the accumulation of accidentals in the final two measures leads to uncertain chord reports, contributing to a wandering key analysis as the piece closes. Here, it is possible for the key agency to make different analyses on different performances. Sometimes the piece comes out in a minor, and sometimes in E major: the instability of the chord reports, caused by ambiguous chords like the penultimate d# diminished seventh, means that the key tracker will be less stable as well. The key analysis is particularly useful in going through the phrase boundary analysis, shown next:

Den	Reg	Speed	Dyn	Dura	Chord		Key	Beat	Total	
1	1	0	0	0	3	I	0	0	5	7
- m2										
0	0	4	0	0	3	i	0	2	9	7 *
1	1	0	0	0	1	ii	4	2	9	8 *
0	1	0	0	0	3	i	0	2	6	8
1	1	0	0	0	1	ii	0	2	5	8
1	0	0	1	0	3	i	0	0	5	8
- m3										
1	0	0	1	0	5	V	0	2	9	8 *
1	1	0	0	0	1	ii	0	0	3	8
1	1	0	0	0	3	i	0	2	7	8
1	0	0	1	0	2	IV	0	0	4	8
1	0	0	0	0	5	V	0	2	8	8
1	1	4	1	0	1	ii	0	0	8	8
- m4										
1	0	4	0	0	3	i	0	2	10	8 *
1	1	4	0	0	2	iv	0	0	8	8
1	1	4	0	0	5	V	0	0	11	8 *
1	1	0	0	0	5	V	0	0	7	9
1	1	0	0	0	0	bVI	0	0	2	9
1	1	0	0	0	0	bVII	0	2	4	9
1	1	0	0	0	5	v	0	0	7	9
0	0	0	0	0	1	biii	0	2	3	9
1	1	4	0	0	2	iv	0	0	8	9

	Den	Reg	Speed	Dyn	Dura	Chord		Key	Beat	Total		
-	m5											
	1	1	0	0	0	5	V	0	2	9	9	
	1	1	4	1	0	6	I	0	2	15	9	*
	0	1	0	1	0	5	V	0	2	9	9	
	0	1	0	1	0	5	V	0	2	9	9	
	1	1	0	0	0	6	i	0	0	8	9	
	0	1	4	1	2	2	IV	0	0	10	9	*
-	m6											
	0	0	4	0	2	2	IV	0	0	8	9	
	0	0	0	1	0	0	vi	0	2	3	9	
	0	1	0	1	0	5	v	0	2	9	9	
	1	0	0	0	0	1	II	0	0	2	9	
	1	1	0	1	0	3	I	0	2	8	9	
	1	0	0	1	0	5	v	0	2	9	9	
	0	0	0	0	0	1	VI	0	2	3	9	
-	m7											
	1	1	0	0	0	5	v	0	2	9	9	
	0	1	0	0	0	0	bVII	0	2	3	9	
	1	1	0	0	0	0	bVII	0	2	4	9	
	0	1	0	0	0	0	bVII	0	2	3	9	
	1	1	0	1	0	1	II	0	2	6	9	
	0	1	4	0	2	3	i	0	0	10	9	*
	1	0	4	1	0	3	i	4	0	13	9	*
-	m8											
	1	1	0	0	2	0	bII	0	0	4	10	
	1	1	0	0	0	3	i	0	2	7	10	
	1	1	0	0	0	3	I	0	2	7	10	
	1	1	0	1	0	1	ii	4	2	10	10	
	1	1	0	1	0	1	III	0	2	6	10	
	1	1	0	0	0	3	i	0	0	5	10	
	1	0	0	0	0	1	ii	4	0	6	10	
-	m9											
	0	0	0	0	0	0	bIII	0	0	0	10	
	1	0	0	0	0	0	bIII	0	2	3	10	
	1	1	0	0	0	3	I	0	2	7	10	
	0	1	0	0	0	3	i	0	0	4	10	
	1	1	4	0	2	3	i	4	0	15	10	*
-	m10											
	1	0	4	0	2	5	V	0	2	14	10	*
	0	0	0	0	0	6	i	0	2	8	11	
	1	0	0	1	0	0	bIII	0	0	2	11	
	1	0	0	1	0	2	iv	0	2	6	11	
	1	1	0	1	0	5	v	0	0	8	11	
	1	0	0	0	0	1	bvii	0	2	4	11	
	1	0	0	1	0	4	III	0	0	6	11	

	Den	Reg	Speed	Dyn	Dura	Chord		Key	Beat	Total	
-	m11										
	1	0	0	0	0	2	IV	0	2	5	11
	0	0	0	1	0	5	V	0	0	6	11
	0	1	0	0	0	0	iii	0	2	3	11
	0	1	0	0	0	3	I	0	2	6	11

Fig. 30

The phrase boundary trace should be read as follows: the first five columns report the weights added by featural discontinuities between events. When one of these columns has a non-zero value, the corresponding feature has changed classifications between the event on that row, and the preceding event. Then, in the column marked *chord*, the weight added for chord function is reported. The function reported for each event is also noted. Next, key changes show up as an additional 4 points. Events on the beat are awarded an additional 2 points, shown in the next column. The next to last column is the total of the activation from all the previous ones, and the number in the far right column is the current phrase boundary threshold. Therefore, if the value in the second to last column for any row is greater than the last number (the threshold) a phrase boundary was reported for that event. I have added asterisks to the trace by hand to mark the identified phrase boundaries.

In the phrase detection for the Bach example, notice that the phrase threshold is initially too low. There are double hits, and spurious boundaries, such as the first two events of measure two. The phrase agency notices this as well, and we see the threshold moving up with each double hit. By the time the threshold gets to about 9, better boundaries are being found. The beginning of the phrase on the last beat of measure 5, for example, is found correctly. The following phrase beginning, at the end of measure 7, is also correctly identified. Here we see the double-hit problem arise again, however. Evidence for phrase boundaries in this scheme seems to accumulate, and dissipate too slowly, leading to identifications of phrase group boundaries on neighboring events. The final phrase boundary in the example, at the end of measure 9, shows exactly the same behavior: the correct boundary is identified, but the following event is chosen as well. This problem can be

easily corrected in the program, as it is, simply by ignoring double hits. The main cause of the problem is the speed change weight. After the fermatas, the program finds the change in duration between events, and contributes to the boundary identification. However, since the speed change at a fermata is so pronounced, successive events tend to each report changes in classification, leading to the double hits.

3.5 Regularity

In the trace of the phrase boundary agency shown above, discontinuities between feature classifications for adjacent events are used as evidence for phrase boundaries. The *regularity* report on level 2 is closely allied to the group detection phase: the classification of regularity and irregularity is done over the history of a group. That is, once a group boundary has been found, regularity/irregularity tracking begins anew. The first event of any phrase, therefore, will always be regular for all features, since it is the only event in the phrase. Once two and more events are present in the current phrase, regularity judgements will again characterize a significant population of events. Then, for each feature, the number of discontinuities between events within the current phrase is calculated, each time a new event arrives. If there are more discontinuities than identities, the feature is said to be behaving irregularly. If feature transitions are identical more often than they are different, the feature is behaving regularly.

In this process, level 2 is using the abstraction reported by level 1, the featurespace word, to characterize the behavior of the music at a higher hierarchic level. The three manifestations of hierarchy in the listener can be seen here: regularity reports concern groups of level 1 events, spread out over time, and depend on the abstractions developed by a lower level in their calculation.

3.5.1 Direction

The complement to regularity classifications is a detector of direction. If feature classifications are changing in a directed manner, the nature of the change should be reported by the listener. For example, if reports from the

register agent are increasing gradually over time, a direction message would qualify the regularity report by noting that the value is increasing with every new classification. The musical phenomena captured by such reports would include things such as decelerando, accelerando, crescendo, decrescendo, etc. In other words, any kind of motion where changing featural classifications present an identifiable direction of change over time.

The identification of direction has not been implemented in *Cypher*; this is the main functionality which could not be produced due to the limited precision of the feature classifications. The problem is easily seen in those features represented by a single bit: change can go only from one state to the other. There is little directed motion to be discerned in an on/off switch. The two-bit classifications are only slightly better. Looking for direction in feature classifications would make sense with more precise reports. In that case, direction characterization would be a natural adjunct to regularity; features could be showing regular behavior within a phrases, but be rising gradually over several phrases. In effect, this points to another level above level 2, looking for change (particularly directed change) spanning many phrases.

3.6 Ensemble Listening

This thesis does not treat the problem of ensemble listening: hearing and making sense of a polyphonic texture of music arising from multiple sources. Though it is not able to process such a situation, the *Cypher* listener could be fit into a framework designed for tracking many sources. To the extent that the program can already make sense of keyboard playing, it is able to deal with polyphonic textures. Most of the analysis done by a single *Cypher* listener would be appropriate to an ensemble, in which case the program would be able to characterize group textures in the same way it does polyphonic keyboard music.

However, a better solution would be to assign a separate listener to each member of the ensemble. Each instrument could then spawn its own response track, in effect doubling the instrumentation. A still better approach would be to add a master listener, looking at the reports emanating from the collection of ensemble listeners. Again, what I am describing is a third-level

listening function: being able to track and characterize the output of several listening streams. The third level functionality described in the direction discussion would deal with groups of phrases; ensemble listening would deal with several simultaneous phrases. This suggests that a third level should be built to deal with at least two different dimensions: characterizing the evolution of several phrases through time, or the collective behavior of several phrases played at the same time.

Chapter 4

Composition

> *As a simple but understandable figure of the imagination, we each have in our minds a committee of "experts" which are the criteria we will consult when making decisions. These criteria are of various kinds: some are inherited, some are needs, but there are also appointed criteria, and there is a time in which they can and will be in this appointed position. If, however, you find repeatedly that this committee doesn't come to a conclusion you actually approve of, you fire it. But then you have to find other criteria. Composition is a wonderful method for discovering not-yet-appointed criteria.* – Herbert Brün [Brün85 pp. 7-8]

In this chapter, I will describe in detail the composition methods built into *Cypher*, looking at their operation, combination, and contribution to complete musical textures. Some of the methods are idiosyncratic solutions to problems arising in my own compositions; others address more widely spread situations in music composition. After cataloging the methods implemented, I will discuss the underlying concept of generating music with interacting agents, how they may be used to approach "expressive" performance, issues of coordinating and combining many real-time composition processes, and techniques for integrating the three main algorithmic styles laid out in the opening sections.

4.1 Levels and Methods

As in the case of the listening processes discussed in the preceding chapter, the composition methods are organized by their level of operation. Methods on the first level operate, generally speaking, on individual sound events. Second level methods are concerned with the direction and regularity of groups of events. This distinction between levels is blurred, perhaps even more so than is the case on the listener side. Level 1 methods often deal with local transitions between groups of two or three notes; level 2 methods could well end up affecting only a single event.

There are three main composing strategies for processes on the generation side: 1) *sequencing*, or the use of prerecorded musical fragments; 2) *transformation*, simple, modular changes to already complete musical material; and 3) *algorithmic generation* from some small collections of pitch or rhythmic elements. The image of a network of processes is the most accurate picture of how these processes interlock; we will see in the course of this chapter how collections of methods can cooperate to generate recognizable features on different musical levels.

4.1.1 Scheduling

All of the compositional methods described in this chapter rely on a scheduler process adapted from [Boynton86]. The scheduler allows timed execution of any procedure, called with any arbitrary list of arguments. A centisecond clock is maintained in the MIDI driver (also adapted from work by Lee Boynton), and is referenced by the driver to timestamp arriving MIDI data. The clock keeps the number of centiseconds which have passed since the MIDI driver was opened. Accordingly, MIDI events are marked with the current clock time at interrupt level, when they arrive at the serial port. This same clock is used by the scheduler, to time the execution of scheduled tasks.

Tasks have associated with them a number of attributes, which control initial and (possible) repeated executions of the scheduled function. We can see these attributes in the argument list to Scheduler_CreateTask, the routine which inserts a function into the scheduler task queue.

```
TaskPtr
Scheduler_CreateTask(time, tol, imp, per, fun, args)
        short imp, tol, per;
        long  time;
        void  (*fun)();
        arglist  args;
```

The first argument is *time*, which is the absolute clock time at which the function is to be executed. Absolute clock time means that the time point is expressed in centiseconds since the opening of the MIDI driver. The *tolerance* is an amount of time allowed for startup of the function. If there is a tolerance

argument, the function will be called at a clock time point which is calculated by subtracting *tolerance* from *time*. This accommodates routines whose effect will be noticed some fixed amount of time after their execution. In that case, the function can be scheduled to take place at the time its effect is desired, and the startup time is passed along as a *tolerance* argument.

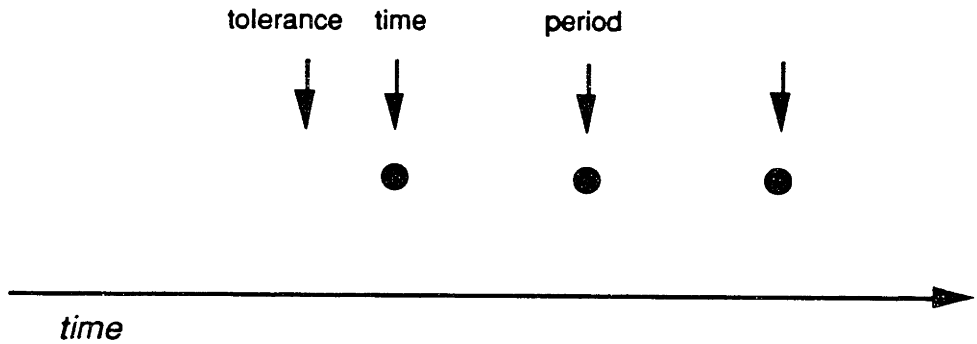


Fig. 31

In Figure 31, the black bullets represent sounding events, produced by some player process. The process needs a fixed amount of time to produce the sound, a *tolerance*, represented by the arrow marking a point some time in advance of the bullet. The call to Scheduler_CreateTask, then, would give the *time* for the desired arrival of a sounding event, with a *tolerance* argument to provide the necessary advance processing.

The scheduler maintains three separate, prioritized queues, and guarantees that all waiting tasks from high priority queues will be executed before any tasks from lower priority queues are invoked. The *importance* argument determines which priority the indicated task will receive.

If a non-zero *period* argument is included, the task will reschedule itself *period* centiseconds after each execution. Periodic functions are conveniently handled by this facility: in *Cypher*, many of the player processes expressing the current beat offset will use the *period* argument. Tasks which are rescheduling themselves automatically can be halted by an explicit kill command at any time. In Figure 31, a *period* argument would continue the sounding events at regular intervals after the first one as shown. The

tolerance argument remains in force for each invocation, providing the advance processing time required. Finally, a pointer to the *function* to be called, and the *arguments* to be sent the function on execution, are listed. The arguments are evaluated at scheduling time, rather than when the function is eventually invoked.

4.2 Transformation

Level 1 methods are invoked by messages arriving from the level 1 listener. They tend to operate on local musical contexts, rather than across spans of several events. However, the level hierarchy is much looser on the composition side than it is for the listener. All of the composition methods shown on the interface look the same to the program; that is, they are interchangeable in terms of their connectivity to listener messages. The arrangement on the interface enforces a hierarchical separation of composition methods, which I will follow in describing them here. After an initial exposition, possible networks of generation processes mixing several levels of control, will be considered.

The composition methods available on the interface for connection to level 1 listener messages are *transformations*, small processes which perform a consistent, recognizable modification to some musical material arriving at their input. The material which is thus modified comes either from the outside world, or, if the program is performing introspectively, from *Cypher* itself.

4.2.1 Note and Event

Before proceeding, let us examine some of the structures used by the program to realize composition methods. The *Note* and *Event* structures are two of the most important ones. A Note closely follows the MIDI standard's concept of a note: it records a pitch (using the MIDI pitch numbering system, where middle C = 60) and a velocity, representing the strength with which the note is attacked. *Cypher* adds a *duration* to these parameters, where the duration is length of time in centiseconds between the MIDI Note On message which initiates the note, and the Note Off message which terminates it.

```
typedef struct {
        unsigned char pitch, velocity;
        long duration;
        } Note;
```

An *Event* structure holds at least one *Note*. Events are kept in doubly-linked lists, which facilitate navigation forward and backward through temporally proximate events (the progressive perspective). The *prev* and *next* pointers point to the preceding and succeeding Events, respectively.

The most important pieces of information added to the collection of notes held in an event are timestamps, which associate an event with a point in time. There are two ways of identifying the onset time of a note: the first records the absolute *time* of the event relative to the beginning of program execution. The second method records the time *offset* of the event relative to the event preceding it. It is this second timestamp which is most often used; in fact, the absolute timestamp is used mainly to calculate the relative offset, and the duration of Notes. The *chordsize* indicates how many Notes an Event contains, and the *data* array holds the Notes themselves.

```
typedef struct ev {
        struct ev *prev, *next;
        long    time, offset;
        char    chordsize, external;
        Note  data[NOTEMAX];
        } Event;
```

4.2.2 Method and Link

As noted earlier, all composition methods look the same to the program: their division into hierarchical layers is a function of the layout of the interface, and a conceptual division of processes dealing with individual events, and those dealing with groups of events. Their implementation in *Cypher*, however, provides identical handles for the connection to listener messages: different interfaces, or other orientations to the grouping of methods, could use the same representations to realize other patterns of

response. Now we will examine the actual data structures which handle the invocation of methods, and their connection to listener messages.

Methods are processes on the composition side that are invoked in response to messages coming from the listener. The data structure Method represents these processes. The primary field in a Method is a pointer to the routine which actually does the work. A Method also has a character string *name*, which will be used to identify it on the interface. The listener messages (features, regularity, phrase boundary) attached to the method are recorded in a bitmapped *callers* word, so that the method may be unconnected from all its callers.

There are two numerical identifications of a method : the first is an *index*, which does not change, and therefore always uniquely identifies a method; the second is a *priority* number, which is used to order the sequence of method executions, and which may be changed by the user. The priorities are reflected on the interface by the ordering of the methods on the screen: methods are executed from right to left, so the higher the priority of the method, the closer to the left edge of the screen it will appear.

```
typedef struct {
        unsigned long callers;
        char index, priority;
        char name[5];
        short (*routine)();
        } Method;
```

A *Link* is the structure which holds connections between listener messages and composition methods.

```
typedef struct {
        char name[5];
        short number[2];
        Method *Methods[2][NUMFILTS];
        } Link;
```

When a connection is made between a message and a method, changes are made to both the Method structure on the composition side, and the Link

structure on the listener side. The structure for the composition method gets the number of the calling message mapped into the *callers* word; the Link structure representing the listener message has its *number* incremented, and a pointer to the new method added to its *Methods* array. There are two *number* and *Methods* fields for each Link: one corresponds to methods connected by an OR operator, and the other to methods connected with a NOT operator. Recall that methods stored in the OR list will be executed when the associated feature or regularity is asserted; methods in the NOT list will be executed when the same feature or regularity is negated. Each Link again has a *name*, identifying it on the interface.

4.2.3 Level 1 Filters

The transformation modules implemented in *Cypher* all accept the same three arguments : 1) a *message*, which selects one of two possible methods in the module; 2) an *event block*, a list of up to 32 events which may be transformed; and 3) an *argument*, whose function changes according to the message. The two messages accepted by transformation modules are xform and mutate. The xform message selects the method which applies a transformation algorithm to an *argument* number of events in the *event block*. The mutate message will use the *argument* to change the value of some parameter controlling the behavior of the transformation algorithm. In this case, the *event block* is left untouched. All transformation modules return a value, when called with the xform message, which represents the number of events in the block to be output after the transformation was applied.

Often, several transformations will be applied to the same event block. In this case, the transformations will be executed serially, with the output of one transformation sent through to the input of the next. In this situation, it makes a difference in which order the transformations are applied. For example, the modules *arpeggiate* and *grace* will produce two different effects, depending on which of the two is applied first. If applied as *grace -> arpeggiate*, a quick grace note pattern will lead up to an arpeggiation of some input chord. In the *arpeggiate -> grace* order, the arpeggiation would be done

first, and then all notes in the arpeggio would have a separate grace note figure added to them.

Because of this difference, each transformation filter has associated with it a priority. Before applying a series of filters, the composition section will order the filters to be used by their priority index. Then, the filters will be executed serially, with the highest priority transformation performed first, the result of this sent on to the next highest priority transformation, and so on. The user can change the priority of any filter through a simple manipulation of the interface. The association of priorities to filters is remembered from execution to execution as part of the program's long-term memory. Further, priority orderings can be changed in performance as different reaction types demand different application sequences.

Following are descriptions of the level 1 transformation modules. The descriptions will be in terms of how the module changes the event block arriving as input, and what kinds of changes to the transformation behavior can be effected by incoming mutate messages.

• **Accelerator.** The *Accelerator* shortens the durations between events. *Cypher* events have associated with them a value called an *offset*, which is the duration in centiseconds separating it from the previous event. Shortening this value causes an event to be scheduled for execution sooner than would be the case were the offset left unaltered; this quickening of execution time results in the events being performed at an accelerated rate.

The state variable controlling the behavior of the accelerator is called *downshift*. In the normal case, *downshift* is just the number of centiseconds subtracted from every event offset in the block. A mutate message can be used to change the value of *downshift*. Increasing it will cause events to be scheduled more quickly; decreasing the downshift value will slow down event scheduling.

The acceleration algorithm is applied to all events in the input block except the first. The first event in every block has something of a special status: normally, the first event will be performed with no delay. This is because we

generally want the response to be as fast as possible. Indeed, a transformed event with no delay will be virtually simultaneous with the original input. To achieve the greatest responsiveness, the first event in a block is given an offset of zero, before all transformations. Accordingly, the accelerator has no effect on the first event (unless some other transformation has already replaced its offset with a positive value).

• **Accenter.** The *Accenter* puts dynamic accents on some of the events in the event block. There are two state variables associated with it: a *count*, which keeps track of how many events have been processed; and the variable *strong*, which indicates how many events are to pass before an accent is generated. The effect of the module is to place a dynamic accent once every *strong* events, which often will induce a metric interpretation in which the strong events act as downbeats. A mutate message can be used to change the value of *strong*; using it, higher level processes can change the accentuation pattern. An accent is described in terms of the MIDI velocity scale, in which 127 is the loudest onset amplitude. The *Accenter* sets the velocity of strong events at 126, and of other events (to highlight the accentuation) at a significantly lower value.

• **Arpeggiator.** The *arpeggiation* method unpacks chord events into collections of single-note events, where each of the new events contains one note from the original chord. The algorithm is straightforward: for each note in the input event, a new event is generated, with the pitch and velocity taken over from the original chord note. The state variable available for mutation is *speed*, which determines the temporal offset separating the events of the arpeggio. Arpeggiated events will be scheduled to play at intervals of *speed* centiseconds; therefore, increasing the speed variable will slow down the succession of arpeggiated notes, and decreasing the speed variable will quicken them.

• **Backwards.** The *Backwards* module takes all the events in the incoming block and reverses their order, so that the event which would have been played last will instead be output first, and so on. There is no mutation message provided for *Backwards*.

• **Basser.** *Basser* plays the root of the leading chord identification theory, providing a simple bass line against the music being analyzed. The state variable *sensitivity* is available for mutation in this routine. Sensitivity refers to the confidence rating being returned from the chord identification agent for the current theory. Each time *Basser* is called, it queries the chord identifier twice: once to get the current root, and again to read the confidence level. If the confidence level is higher than the value of *sensitivity*, a bass note will be played; otherwise the module remains silent.

Root notes are played within the second full octave on a piano keyboard, that is, within the MIDI pitch range 36 to 47. Further, new bass notes will be played no faster than one every 25 centiseconds; it is usually hard on synthesized bass sounds, and somewhat disconcerting, to hear a bass line with attacks several times a second. *Basser* does not funnel its output through the event block mechanism, but schedules its own performance. This is because there is no transformation of the input being performed; a new voice is being generated in addition to the original events.

• **Chorder.** The *Chorder* module will make a four-note chord from every event in the input block. There is an array of three intervals, which will be used to limit the pitches in the chord. For every event in the block, the first pitch in the event is taken as a starting point, and the other three pitches are generated as intervals within the bounds given by the interval array. For example, the first pitch is chosen at random within seven half-steps of the original note. The second must be within 15 half-steps, and the last within 23. If any pitches exceed the upper limit of the keyboard range, they are changed to the highest pitch on the keyboard. Using this algorithm, all chords generated by the *Chorder* will be smaller than two octaves in total span, and contain four pitches more or less evenly distributed throughout that range. There is no mutation message available for this module.

• **Decelerator.** The *Decelerator* lengthens the duration between events. Increasing an event offset causes it to be scheduled for execution later than would be the case were the offset left unaltered; this slowing of execution time results in the events being performed at a decelerated rate. The state variable affecting the behavior of the decelerator is the *upshift*: the number of

centiseconds added to every event offset in the block. A mutate message can be used to set the upshift to a new value. Increasing the upshift will cause events to be scheduled more slowly; decreasing the downshift value will speed up event scheduling.

For every event in the input, *upshift* is added to the offset, if that lengthening results in a value less than 400 centiseconds. If the new offset exceeds that threshold, in other words if the resulting offset would cause the event to be delayed by more than 4 seconds, 200 centiseconds are subtracted from the offset instead. This will cause the event to be scheduled two seconds earlier, in effect an acceleration; however, this quickening also means that subsequent decelerations will again cause a slowing within the 4 second margin. The overall behavior will be a gradual slowing, to the 4 second limit, one quick acceleration, and another gradual decelerando.

• **Flattener.** The *Flattener* is one of the few modules which will undo variation found in the input block. This module flattens out the rhythmic presentation of the input events, setting all offsets to 25 centiseconds, and all durations to 20 centiseconds. The result is a performance which is significantly more machine-like than most inputs, since all attacks will be exactly evenly separated one from another. There is no mutation variable associated with this transformation.

• **Glisser.** The *glisser* adds short glissandi to the beginning of each event in the input block. There is one state variable available for mutation, which controls the maximum length of the generated glissandi. The first step of the transformation method calculates a glissando length as follows: `howmany = ((rand()%length)+1);` The random number generator is called, and its output used modulo the *length* variable, to limit the upper bound. One is added to the result, to ensure that all executions will add at least one new event to each incoming event. Therefore, mutating the *length* variable will change the maximum number of notes allowed in a glissando.

Glissandi are generated from below the input note, and run up to it in half-step increments. The interval between the input event and the first note of the glissando, therefore, depends on the calculated length. If, for instance, a

glissando length of five is needed, the first pitch will be a perfect fourth below the input event, to allow five half-steps of ornamentation going up to the original pitch. These new pitches are generated with a constant speed (7 centiseconds apart) and velocity (rather loud – MIDI value 110).

• **Gracer.** The *Gracer* appends a series of quick notes leading up to each event in the input block. Every event which comes in will have three new notes added before it. All of these new notes will be at offsets of 10 centiseconds, resulting in a fast, grace-note like ornamentation of the original event. The added pitches are chosen at random from a range set by the *space* variable. Grace notes will usually be chosen to appear below the original pitch; random numbers are generated within the *space* range, and then subtracted from the event pitch. If this results in a note below MIDI number 40, the modification is instead added to the original pitch, producing a grace note figure which leads down to the original event. The *space* variable can be modified with a mutate message, changing the pitch range within which grace notes will be selected.

• **Harmonizer.** The *Harmonizer* modifies the pitch content of the incoming event block to be consonant with the harmonic activity currently active in the input. The basic idea of the algorithm is the following: after ascertaining the current chord and key, pitches in the event block are nudged into a scale consonant with those analyses. The contour of the pitches is maintained as closely as possible; pitches dissonant with the chord and/or key are moved up or down one or two half-steps to a consonant pitch. For example, an f sharp found in an event block associated with the tonic chord in C major would be nudged up to a g natural, changing a highly dissonant pitch to a consonant member of the tonic triad.

The specifics of how to modify incoming pitches are maintained in the *nudge* array, which has three dimensions, corresponding to mode, function, and original pitch. The first dimension, *mode*, has two possible values, which represent major and minor versions of the current key. This allows an immediate selection between two kinds of modification, one for each mode. The second dimension goes into the mode-specific transformations and finds the list associated with the *function* of the current chord. Function is used in

the music theoretic sense here: a C major triad in the key of C major, for example, has a tonic (or, in the program's numbering scheme, zero) function. There are 24 possible functions, corresponding to the major and minor versions of triads built on the twelve scale degrees. The final dimension of the *nudge* array comes from the pitch number of the incoming note, which is considered regardless of octave, meaning that the value of this dimension can be one of twelve possibilities.

With these three dimensions, the harmonizer is equipped to transform any incoming pitch class, played against any chord function in either the major or minor modes of the key. There are no mutation possibilities associated with this module, but should be: according to the style of music being performed, the *nudge* array itself should be replaced. Some styles place stricter constraints on allowable dissonances, and the way these are resolved, than others. If the program were able to recognize styles, and had at hand a repertoire of *nudge* arrays corresponding to them, it could swap in the appropriate harmonic behavior for each known style.

• **Inverter.** The *Inverter* takes the events in the input block and moves them to pitches which are equidistant from some point of symmetry, on the opposite side of that point from where they started. In other words, all input events are inverted around the point of symmetry. For example, if the symmetry point were middle C, all pitches above middle C would be transformed to others the same distance below middle C as the original pitches were above it. High becomes low, and the higher the original pitch, the lower the transformed pitch.

The point of symmetry can be altered with a mutate command: the default setting is 64, which is the MIDI pitch number for the E above middle C, the exact center of an 88-key piano keyboard. A mutation can change this point to any other MIDI pitch number. Pitches which, after transformation, would exceed the MIDI pitch boundaries, are pinned at either the highest or lowest possible pitch, whichever is closer to the calculated value.

• **Looper.** The *Loop* module will repeat the events in the input block, taken as a whole. In other words, all of the events in the block will be performed once,

then all performed again, and so on, until the desired number of loops is reached (in contrast to a looping scheme in which each individual event would be repeated some number of times before continuing on to the next).

The number of loops performed depends in part on the *limit* variable. There are always a minimum of two loops performed, including the original events. So, the least action taken by this module would be to repeat the original events once. Additional repeats are generated at random, up to the value of *limit*. For example, if *limit* were equal to 4, the module would perform randomly from one to five repeats. The *limit* can be reset with a mutate message.

• **Louder.** The *Louder* module adds crescendi to the events in the input block. This is another case in which the transformation would not be heard on blocks of only one event. The solution here is to add a second event to singleton blocks. The amount of velocity change is arrived at quasi-randomly; that is, the following calculation is used: change = (rand()%limit)+1; This varies the velocity modification randomly up to a maximum set by the variable *limit*, with a minimum value of 1.

Each event in the block will receive an increasing augmentation of its velocity. The first event has *change* added to it. Before each succeeding event, *change* is increased, then added into the next event velocity. If ever the new velocity exceeds the MIDI maximum (127), the value of *change* itself is used. Then, the algorithm proceeds as before. The effect is a crescendo, more or less gradual (depending on the value of *limit*), up to the MIDI limit, followed by a sudden drop in loudness, followed by another gradual crescendo. A mutate message can be used to change the value of *limit*, effectively varying the speed of loudness change.

• **Obbligato.** This module adds an obbligato line high in the pitch range to accompany harmonically whatever activity is happening below it. The algorithm for doing this is very simple : the chord agency is queried for the root of the current harmonic area. The root is played out in the octave starting two octaves above middle C. The other wrinkle on this behavior is that the module only plays out an obbligato note for every fourth event in the

incoming event block. This ensures that the obbligato line will move more slowly than the other material. The frequency, in events, of obbligato output can be changed by sending the module a mutate message with a new setting.

• **Ornamenter.** *Ornamenter* adds small, rapid figures encircling each event in the input block. Two new events are added for each one coming in. The new events will circle the original pitch, with one new event above it, and one below. The distances above and below the original pitch are chosen at random, within a boundary controlled by the *width* variable. The calculation is : change = ((rand()%width)+1). *Width* keeps the output of the random number generation within a boundary; one is added to make sure that all new pitches will differ from the original by at least one half-step. The new events have a constant offset (100 msec) and velocity (110 MIDI). The width variable can be changed with a mutate message.

• **Phraser.** The *Phraser* module temporally separates groups of events in the input block. Adding a significant pause between two successive events plays on one of the strongest grouping cues, and tends to induce a phrase boundary at the break. The state variable *phrase_length* determines how many events will pass before a pause is inserted. When *phrase_length* events have gone by, the offset of the following event will be lengthened at random by a duration somewhere in the range of 400 to 1670 msec. The value of *phrase_length* can be changed with a mutate message.

• **Quieter.** *Quieter* adds decrescendi to the events in the input block. Again, the effect would not be heard on blocks of only one event, so a second event is added to singleton blocks. The amount of velocity change is arrived at quasi-randomly; that is, the following calculation is used: change = (rand()%limit)+1; This varies the velocity modification randomly, with a maximum somewhere below the value of *limit*, and a minimum of 1.

The value of *change* is subtracted from the velocity of the first event in the block (which have become two if the original input was a single event). For each succeeding event, first the value of *change* is augmented, and then the event's velocity is diminished by that value. If ever the newly calculated velocity goes below 10, *change* is subtracted from 127 (the maximum MIDI

velocity), and the result used as the new event velocity. The effect will be a gradual decrescendo (the speed of which is determined by *limit*), an abrupt jump to a loud velocity, and another gradual descent. The value of *limit* can be changed with a mutate message, varying the speed of loudness change.

• **Sawer.** This module adds four pitches to each input event, in a kind of sawtooth pattern. It is similar to the *Ornamenter*, in that it also adds new material above and below the original pitch. In this case, two pitches will be added above, and two below, in alternation. The amount by which these new notes will deviate from the input is determined by the state variable *width*.

The calculation for finding the interval between the input event and the sawtooth pitch is : `n->pitch += (((rand()%width)+1)*((j%2)?1:-1));` Here we can see that the value of *width* will limit the output of the random number generator. One is added to the result of that operation, to make sure that each new pitch will differ by at least one half-step. Finally, the counter *j* is used to find the direction of the interval: odd values of *j* will produce pitches above the original event, and even values produce pitches below. The variable *width* can be changed with a mutate message.

• **Solo.** The *Solo* module is the first step in the development of a fourth kind of algorithmic style, lying between transformative and purely generative techniques. Because its operation is not a transformation, in the same sense as the other processes described in this section, I will reserve discussion of *Solo* for section 4.5.

• **Stretcher.** The *Stretcher* affects the duration of events in the input block, stretching them out beyond their original length. The state variable *mod* controls the range of variation which will be introduced. The calculation of the new duration for each event in the block is coded thus: `duration = (((long)(rand())%mod)+50L);` First, a random number is generated; this is limited to the maximum specified by *mod*. The result of the modulo operation is added to 50, giving a minimum lengthening of 50 centiseconds. The maximum lengthening is determined by *mod*, and the value of *mod* can be changed with a mutate message.

• **Swinger.** *Swinger* modifies the offset times of events in the input block. The state variable *swing* is multiplied with the offset of every other event; a value of *swing* equalling two will produce the familiar 2:1 swing feel in originally equally spaced events. If the input events are not equally spaced to begin with, the swing modification will have more complex results. The value of *swing* can be changed with a mutate message.

• **Thinner.** The *Thinner* reduces the density of events in the input block. As we have seen, most of the transformations, if they change the number of events presented at the input, change it by adding new events. Consequently, the density of events emanating from the composition section can reach quite significant levels. *Thinner* is one tool for reducing the amount of material coming from the player.

The state variable *thin* controls how the reduction is done. A count is associated with the module, which is incremented with each note of every event. When *thin* divides evenly into the count, the corresponding Note is deleted, and the offset of the Event which contains the Note is increased by one second. For example, if *thin* were 3, and an incoming event held six notes, two of the notes would be deleted, and the offset of the event increased by two seconds. The density of incoming material will be reduced, then, in two ways: some chords will hold fewer notes, and pauses of one or more seconds will be added between some events. The value of *thin* can be changed with a mutate message.

• **TightenUp.** To do the *TightenUp*, events in the input block are aligned with the beat boundary. On entering the module, the beat agency is queried for the current beat period. Next, the offset times of all events in the input block are added together. If the combined offsets are equal to the beat period, then the final event of the block will sound on a beat boundary. This is the desired effect of the module: if the offsets are already in such an alignment, the method returns with no further action.

If the combined offsets have a duration less than the beat period, the last offset is extended by the difference between the two – again placing the final event in the block on the beat. If the combined offsets' duration is longer than

the beat period, the difference is divided by the number of events in the input block, and the result is subtracted from every event offset. In other words, all events are quickened by a constant value to make the final event again fall on the beat. There is no mutate message associated with this module, and the transformation does not add any events to the ones already present on input.

• **Transposer.** *Transposer* changes the pitch level of all the events in the input block by some constant amount. The distance by which the pitches will be moved is calculated as follows: interval = (base%limit); *base* is a random number; *limit* will keep the value of this random number within some bound; and *interval* is set to the result. Then, for each event in the block, *interval* is added to the current pitch. If the resulting pitch exceeds the upper limit of the pitch range, interval is instead subtracted from the original event pitch. So, transpositions will generally be upward, though near the top of the pitch space they will move the other way. The value of *limit* can be changed with a mutate message.

• **Tremolizer.** The *tremolo* module adds three new events to each event in the input block. The new events have a constant offset of 100 msec. They surround the original pitches, with two new events above the original, and one below, or two below and one above; in either case, the higher and lower new events alternate. The distance new pitches will be removed from the original is determined by the following calculation: short width = rand()%12+2; First a random number is generated. This is limited to a maximum of 12, then added to two. The addition ensures that all new pitches will differ from the original by at least one whole step. The maximum deviation between original and new pitches is an octave plus one whole step. There is no mutate message associated with this module.

• **Triller.** *Triller* adds four new events to each event in the input block. These will performed as a trill above or below the original pitch. The new events have a constant offset of 100 msec. The placement above or below the source pitch is determined randomly, but trills which would come out below MIDI pitch 30 are always played above the source, and trills which would come out above MIDI pitch 100 are always played below. The trills will be the source pitch alternating with a trill pitch either a half-step, or a whole step, above or

below it. The entire figure will begin and end on the source pitch, alternating with two trill notes. The choice between half or whole step trills is made randomly. There is no mutate message associated with this module.

4.2.4 Level 2

Level 2 composition processes are invoked in response to messages from the level 2 listener, which communicate patterns of regularity, grouping, and direction. Level 2 composition similarly is concerned with behavior over groups of events, and often is accomplished by changing the behavior of level 1 composition processes over time. One way to effect such change is to modify the connections between level 1 features and their associated transformations, and this is a common strategy in level 2 methods. Another way to change the compositional behavior of level 1 is to send mutate messages to the transformation modules.

We have seen in the previous discussion the kinds of mutation which can be generated by changing the internal state of the transformations. Using either of these strategies, along with some others, level 2 methods can affect composition at the phrase level, in response to the regularity or direction of features within the phrase, or at boundaries between phrases.

Following are descriptions of implemented level 2 composition processes:

• **VaryDensity**. Used to establish and break level1 connections between the density features and certain transformations. It will be invoked by whichever level2 listener message is connected to it; then the process examines the featurespace classification to determine if the current input is a chord or single note. If it is a chord, a level1 connection will be drawn between the chord feature and the arpeggiation transformation. If the input is a single note, a level1 connection will be drawn between the line feature and the chord transformation. In other words, the density of the input to the listener will be reversed in the *Cypher* composer's output.

• **UndoDensity**. Disconnects all level1 methods from the vertical density features, so that no transformations will be invoked from the *line* or *chord*

classifications. This method can be used, for example, to cancel the effect of *VaryDensity*, described above. Regular behavior of some feature could be used to call *VaryDensity*, establishing density-activated transformations on level 1. Irregular behavior of the same feature might be tied to *UndoDensity*, causing all level 1 density-activated transformations to cease.

• **Phrase**. Primarily used as a way to mark phrase boundaries. When invoked, the routine sends out a loud MIDI event, and adds a significant delay to the offset of an event block about to be played, emphasizing a potential phrase boundary. The method is designed to be called by the *Phrase* method on level 2; that is, whenever the phrase agency detects a phrase boundary, a message identifying the boundary will be sent out. If *Phrase* is hooked up to that message, it will announce the arrival of the boundary with an audible bang.

• **JigglePitch**. This method sends out a single event, containing one pitch placed randomly anywhere in the keyboard range, every seventh time it is invoked. All other invocations will have no effect. *JigglePitch* is most often used for "jiggling" composition methods that may be stuck in a repetitive pitch pattern or area, and so can be profitably connected to level 2 listener messages denoting regularity.

• **MakeBass**. *MakeBass* establishes a connection on level 1, between the *All* message, and the *Bass* transformation module. Connecting a player process to the *All* message means that the player process will be called for every input event, no matter what its features. As described in the transformations section, the *Bass* module queries the chord agency for the root of the current harmonic area, then plays out a single event, containing the root pitch, in the octave two below middle C.

• **BreakBass**. The *All* message on level 1 is disconnected from the *Bass* transformation module. *BreakBass* acts as the complement to *MakeBass*, undoing the effect of that module. For this reason, the *Bass* module is only disconnected from *All*; any other level 1 features linked to Bass will remain in force.

• **BeatPlay**. This method looks at the period of the current beat theory, and schedules bass notes to be played on the beat. If input is arriving, the beat period will be constantly adjusted to match the most recent theory. If no more external information is arriving, bass notes will be continued, using the period from the most recently calculated beat theory.

• **BeatStop**. The complement to *BeatPlay*, this method will stop any currently scheduled beat task from continuing. Again, these methods are paired, and are often hooked to opposite poles of irregular/regular messages for some feature. For example, a report of regular speed could trigger the beat player, and a subsequent report of irregular speed could be attached to *BeatStop*, causing all beat activity from the player to stop.

• **AccMutate**. This is the first of a series of mutator processes, which take advantage of mutation facilities built into the transformation modules, described in the previous section. *AccelMess* sends a mutate message to the *Accelerator* level 1 module, to vary the rate of acceleration performed. The new values sent will follow the form of a ramp: increasing linearly to a maximum of around 450 (which will be the number of msec subtracted from event offsets by the *Accelerator*), jumping back down to 100, and increasing slowly from there again.

• **SawMutate.** *SawMutate* is another mutation process, this time sending messages to the level 1 *Sawer* module. As we saw in the previous section, *Sawer* will change the width of sawtooth-like ornamentations in response to mutate messages. *SawMutate* changes this width randomly, up to a maximum of two octaves.

4.3 Compositional Algorithms

Like transformations, what I call *compositional algorithms* can be invoked in response to any of the messages coming from the listener. The algorithms exhibit a variety of distinguishable styles, and themselves take additional messages to influence their behavior. The initial invocation of a compositional algorithm is accomplished with a SET message, which will initialize internal variables, and establish the duration over which the

algorithm will continue to run. Another message known by all of the algorithms is CONTINUE, which causes another invocation of the same routine to be scheduled, continuing the algorithm's generation.

• **Tremolo.** The *tremolo* algorithm produces a rapidly changing pitch field which sweeps up the keyboard range, folding around to the bottom of the range when it has reached the top. The overall duration of the process is established with the set message. Within that duration, the algorithm calls itself many times, using continue messages. The temporal offsets between these self-invocations are determined by a calculation which alternately lengthens, then shortens the duration between calls.

```
/* vary pct at unsymmetrical rate (fast up, slow down) */
if (inc > 0.0) {
        if ((pct+=inc) > 187.0) inc = -09.0;
} else {
        if ((pct+=inc) <  83.0) inc =  19.0;
}

/* calculate onset of next call */
calc = 79.0*(100.0/pct);
```

Each invocation of the routine includes a seed pitch, and a fixed set of three intervals determines the notes generated around this seed. Every seed will produce ten new pitches, arrived at by continually cycling through the array of intervals. Three and 1/3 repetitions of the interval array are heard for each invocation of the routine. The seed pitch itself, then, is shifted upward with each new call of the routine, until it reaches the top of the range, at which point it is folded back around to the bottom. The entire process continues until the duration specified in the set message is met.

• **Turang.** The *turang* algorithm produces a series of chords, moving up and down around a central area. Rhythmically, the chords speed up and slow down progressively, with occasional pauses separating groups of events. The intervals used for each invocation are constant, creating a pattern of shifting parallel chords. Again, the set message determines the duration of the entire generation. The midpoint of the pitch area within which material is to be

generated is also initialized with set. Then, each subsequent invocation of the routine with a continue message will include a new seed pitch generated somewhere within two octaves of the initialized midpoint. Even calls will use a seed pitch below the midpoint, and odd calls a seed above it. Therefore, the behavior of the algorithm will exhibit an alternation in pitch space around the specified central point.

The choice to extend the duration of a given event, and insert a pause following it, is made by generating a random number and dividing it by seven. If seven divides evenly into the random number, a pause is inserted. In this way, there is on average a one in seven chance that a pause will be added.

These two examples typify the set of compositional algorithms implemented in *Cypher*. The strategy behind their construction is to fashion rather simple generators of recognizable textures, which can be parameterized to accord with the features of an ongoing musical context. These textures can be used on their own, or as the seed material for combination compositional processes, discussed in section 4.7.

4.4 Sequencing

Recorded groups of MIDI events are commonly referred to as *sequences*, and the process of recording and playing them back, *sequencing*. Sequencing is little used in *Cypher*, primarily because the emphasis has been on the algorithmic techniques of generating an appropriate response in real time, rather than calling up a prerecorded response under certain conditions. The capability of using sequences in this way, however, is supported by the software, and the idea of launching fragments from a small library of possibilities when associated structures are encountered in the input is part of the orientation of the enterprise. The main implementational concern is that processing of the sequences should not degrade the performance of other listening and compositional tasks.

Sequencing was better supported in an earlier version of the program: the current implementation is not far from providing the same facilities. The

basic idea is this: a process which commences performance of a sequence file can be attached to any listener message. Such a process becomes a composition method like any other; the *routine* field in a Method structure would simply point to the sequence-calling process. Once initiated, a pointer to the sequence file is kept in a list of open sequences. Then, in the main *Cypher* loop, the sequence scheduling routine schedule_next_second() will be called.

Schedule_next_second() looks at the list of ongoing sequences each time it is called. For each pointer in the list, one second's worth of sequenced material is read from the file and sent to the scheduler. All sequencer files follow the standard MIDI file format; events in such files are time-stamped with their offset from the previous event (as is done in *Cypher*). Therefore, schedule_next_second() can send file events to the scheduler, adding up successive offsets until one second's worth of events have been read.

When one file has the required duration of events scheduled, the routine goes to the next pointer in the list, until all have been processed. The idea behind this handling of sequences is that *Cypher* makes a complete pass through its main loop at least once a second. If all open sequences have one second's worth of events scheduled each time through, they are guaranteed to continue playing until the next execution of schedule_next_second(). However, the program will never get stuck scheduling all of a long sequence, while other processing waits, because files are never read all at once (unless they are quite short to begin with).

Because scheduling of sequence files is interleaved with all other processing, *Cypher* can handle the performance of several sequences simultaneously, without a noticeable degradation in its ability to respond to other live input. Similarly, the listener can be pointed at a sequence file, and respond to the music in it through the usual listener/player mechanism, even while it is playing back the original sequence itself.

4.5 Expression

One of the most noticeable differences between the musical performances of human and machine players is the expressive variation invariably added by

human musicians. Expression is performed across several parameters, among them onset timing, dynamic, and duration (articulation). Human listeners to such performances are able to separate structural from expressive timing, recognizing a series of eighth notes played with a ritard, for example, for what it is, rather than experiencing a more complicated set of gradually lengthening notated durations [Clarke87].

Studies in performance practice [Palmer88] are beginning to give us good data about the strategies humans use in expressive performance, critical information if machine players are to learn to incorporate such expression in their own performances. Other expressive parameters in human playing include pitch inflections and timbral changes. The implementation described in this thesis ignores timbral information, and does not include continuous controls such as pitch variation. Therefore, those parameters, though highly desirable candidates for manipulation by a computer performer, are left out of consideration here. Rather, the problem of expressivity was explored in connection with two parameters currently available to the system: variations of timing, and loudness.

4.5.1 Time Maps

Everything we know about human performance tells us that the rhythmic presentations of musical events is never metronomically regular. In fact, it appears to be virtually impossible for humans to maintain a clock-like periodicity between events in their performance. Yet, computer performances are quite often characterized by just such rigid, mechanical renditions of rhythmic material. In any case, this inability of human performers is hardly a deficiency; strictly metronomic renditions eliminate one of the most critical sources of interest in musical performance, and a powerful means of expressing a certain conception of the score.

The compositional processes implemented in *Cypher* provide directed, and static, means of temporal deformation. Directed temporal operations perform a linear transformation of the offset times separating events, either lengthening them (decelerando), or shortening them (accelerando). Static operations add smaller, non-directional changes to the length of event offsets.

These possibilities may, as is the case with all compositional methods, be applied on any level, in response to messages arriving from the listener. Level 1 applications will change offsets on a per-event basis; see the transformation modules *Accelerando* and *Decelerando* described in section 4.2.3. Level 2 applications will be invoked in response to regularity or phrase-boundary messages. Therefore, their action will be advanced with the frequency of the appropriate messages coming from level 2; temporal deformations attached to phrase boundaries will be advanced on a per-phrase basis, for example.

The time map technique described in [Jaffe85] provides a general method for coordinating time deformations among several voices of output. The idea is to be able to specify tempo fluctuations in such a way that voices can speed up or slow down independently of one another, and return to synchronization at any desired moment. As an example, imagine using slight ritardandi to set off the cadence of certain phrases: a time map would allow the notation of that phrase as regular (e.g., a series of eighth notes), but force the performance to include delays of slightly increasing duration between successive note onsets as the cadence nears.

We can think of a time map as a description of the relation between structural (notated) rhythms and expressive (performed) ones. As such, maps are applied quite naturally in situations where there is a score, since a score is a structural notation of the rhythmic content. To use time maps in performance-driven systems with no score, those programs must be able to assert goal points in their output, or in some way to segment the material they are producing, such that a time map can meaningfully adjust the time offsets before performing the fragment.

The listener messages and goal assertions implemented in *Cypher* provide just such handles for the live application of time maps. Goals can be asserted either by the pattern matching agency, or by the critic. The pattern matcher can report that it expects a goal based on the ongoing match of one of its known progressions. The critic can assert a goal as the endpoint of some motion it is trying to induce in the output of the player. In either case, once

we have a goal, the program can fit a time map onto the intervening material to produce directed motion pointing to the desired endpoint.

4.5.2 Dynamic Variation

Dynamic variation is another common conveyance of expressive performance. Changes in loudness are used to emphasize structural boundaries, and to highlight directed motion toward some musical goal. Crescendo and decrescendo are clear examples of expressive dynamic variation, but far from the only ones. As in the case of temporal deviations, slight, quickly changing perturbations of the dynamic level are a critical part of an expressive, or even just acceptable, musical performance. But dynamic variations cannot be applied randomly, or by following only local musical constructs. The essence of expression is to use variation in pointing out structural boundaries, major articulations of the composition in progress. Unless some of that structure is present, either laid out by a human user, or found by the program itself, dynamic variation will add unpredictability to a computer performance, but will not enhance the performance of larger musical gestures, as a human performer normally would.

Dynamic variation in *Cypher* can take three forms: crescendo, descrescendo, or a more quickly changing, unstable pattern of change in loudness. These possibilities subsume two kinds of directed motion (louder, softer), and one relatively static pattern of change. Further, these operations can operate simultaneously on different compositional levels. Imagine the case of a series of decrescendi, embedded in a larger, more global crescendo. Or small, local dynamic changes within an overall decrescendo.

The level structure of *Cypher* provides a framework for directing the application of dynamic variation to different structural planes of the compositional output. If a dynamic process is applied on level 1, for example, changes in loudness will occur on a per-event basis (see the *Louder* and *Quieter* transformation modules). Dynamic processes on the second level will be applied as a function of the listener message to which they are connected: a crescendo connected to the phrase boundary message, for example, will affect

output on a per-phrase basis, with each successive phrase somewhat louder than the last.

Assertions of goal points provide another handle, perhaps the most meaningful one, for the articulation of dynamic expression. Goal assertion can arise either from the frame-matching agency, which could predict the arrival of some goal point as the expected continuation of a noticed pattern. Another source of asserted goals is the set of aesthetic productions in the critic, which may attempt to induce some point of culmination in response to a situation arising in the output of the player. In either case, the dynamic agents described above can be applied in such a way that their endpoint (in the case of directed motion) will coincide with the desired goal.

4.6 Compositional Critic

The *compositional critic* is a process and associated rule set that watches and manipulates the output of the composition section. As described in Chapter 2, there are two listeners active in the current implementation of *Cypher*: one analyzing MIDI data arriving from the outside world, and the other looking at the musical output of *Cypher* itself. The critic is a process that responds to messages coming from the second listener; in effect, the critic corresponds to the player in a higher level application of the listener/player pair. It operates as follows: the second ("critic") listener follows the output of *Cypher*'s composition section. The destination of the listener's analysis messages is the critic, which responds to those messages by performing operations on the material generated by the player, before that material is sent out to make sound.

The critic is able to perform several tasks, all centered around monitoring the output of the composition section. First of all, it can produce a trace of the composition section's activity – recording a picture on one of several levels of detail which shows what the player processes are actually generating. Further, the reports generated by the listener provide the foundation for an evaluation of the music *Cypher* produces: to the extent that reports of regularity, grouping, direction, and so on provide meaningful, coherent descriptions of

musical activity, they can be used as handles to examine and critique the compositional skill of the generation section.

4.6.1 Input Sampling

When a MIDI event has been analyzed by the listener, it is copied into a scratch event block for processing by the composition section. According to the connections established between listener agents and composition methods, events in the block will be modified, or new events added. It is possible, particularly during introspection or in communication with another computer program, that events will arrive more quickly than they can be processed. Even more common is the case that the program will generate more data than the sound synthesis gear can produce. For this reason, the program will sample input which is too dense to be treated successfully.

If a very fast collection of events has arrived since the last iteration of the listener/generation loop, all of the events will be sent through the listener (to keep the analysis record accurate), but only the last event will be processed by the composition section. It was found that treating the final event provides a degree of audible interaction which maintains the dialogue between players, even though a larger portion of the incoming information has not been transformed.

4.6.2 Interestingness

An important function of the critic is to find "interesting" material in output of the player for further manipulation by the composition methods. When performing introspectively, *Cypher* often must sample material being generated by the player . Tracking a human performer, the program is almost always able to keep up with the rate of events arriving for evaluation and response. When the program begins to perform through introspection, analyzing and transforming its own output, data can easily begin to arrive with enough density, that sending out transformations of it all would swamp

both the synthesis gear and the comprehension of the listener.[5] Therefore, the critic needs to select events from the wealth of material emanating from the player, which are then analyzed and used to germinate a fresh round of compositional processing.

To select interesting events for further processing, the critic uses the phrase boundary messages coming from the "critic" listener. As we saw in Chapter 3, a listener will identify phrase boundaries in a musical stream by monitoring discontinuities across feature classifications between two events. When discontinuities have arisen in enough features, following the weighting scheme discussed in section 3.4, the phrase agency reports a boundary. The critic listener performs this operation on material coming from the player. Events which the listener identifies as beginning a new phrase are considered "interesting" by the critic, and sent back to the "analysis" listener for another round of evaluation and response.

It may well happen that no phrase boundary is noticed in a block of material. In that case, the critic will select the event with the most differences from its predecessor. Essentially the same criterion is being used: events with many discontinuities relative to the event previous will be marked as interesting. If several events have the same level of difference from their predecessor, the earliest such event will be used.

Already in such a simple application of the critic, we can see that the rules associated with it form, in effect, an aesthetic bias for the program. Declaring events whose feature classifications differ markedly from their predecessor to be "interesting" is no more or less than an aesthetic choice; there is no inherent reason to find any musical event more interesting than any other. Aesthetic biases can be codified in terms of preferred actions to be taken in response to material exhibiting certain kinds of regularity, or feature classifications, or harmonic behavior – in short, preferences can be established for all of the situations reported out of the listener.

---

[5]The same observation holds for any input with a high enough density of events; introspective composition is the most common case, but others, such as input arriving from another computer program, are handled in the same way.

4.6.3 Aesthetic Productions

The rule set associated with the critic forms just such a collection of aesthetic preferences, applied to the incipient output of the player. The rules are expressed in the form of a production system: a set of condition-action pairs, where the condition parts are expressed in terms of logical operations on listener messages, and the action parts consist of transformations to be applied to the musical material. The productions are applied just before the material is sent to the synthesizers. When events are first scheduled, there is no way of knowing how much additional material will be scheduled to be played at the same time. For that reason, whenever the composition section schedules some new output, it is first sent to a routine called Gather(), which will consolidate events destined for the same point in time. Further, Gather() will arrange for the consolidated material to be sent through the critic one clock tick before the time comes for it to be played.

A simple example is the following: the critic will prefer to reduce the vertical density of material being presented at a high speed (high horizontal density). Such a rule can be expressed as:

```
if (FastVal(featurespace) > 2) Thinner(...)
```

The critic first computes the featurespace and regularity words for the event. A set of routines such as FastVal() is available to examine the classifications being returned for any of the level1 or level2 features and regularities. In this case, FastVal() will return the value returned by the level 1 analysis for the current event. If this value is greater than 2, the speed of this event has reached the highest possible rate. In that case, the material will be reduced through application of the transformation module *Thinner*.

The aesthetic productions held in the critic are now static; that is, *Cypher* has a particular style which it will enforce on the music it plays. In a full implementation, the critic's rule set should be swapped in and out as a function of the style of music being played to the system. Rather than having a simple set of rules applying to the output of the program generally, specific stylistic rule sets could be invoked for known genres seen arriving from the outside world.

An extension to the static critic implements this idea: a further set of production rules can be invoked from the interface, or through a command in a score orientation script. When the additional rules are used, several changes are made to the responses stored in the connections manager according to the characteristics of the music being put out by the composition section. This is another way to approach the manipulation of the player: rather than editing the program's output directly, the way the output will be generated in the first place is changed. Possible manipulations programmed into the action part of these productions include making and breaking level 1 connections, changing the sound banks and timbres used, and reducing the length of time the program will wait before beginning to perform introspectively.

4.7 Combination Techniques

In the preceding review of composition methods, I have roughly divided the algorithms presented into three classes: processes of transformation, of generation, or of sequencing. Another possibility, however, is to construct compositional strategies which combine aspects of all three classes; human improvisation certainly arises from a combination of these. The methods employed most by humans and *Cypher* are nearly inverted, in that the technique which is most difficult for humans (immediately adopting and transforming the material of others) is what *Cypher* does best and most often. Human improvisers rely much more heavily on remembered sequences, which can be called up and adapted to different performance situations, and on the manipulation of sets of basic elements (scales, rhythmic patterns).

In this section, I will first describe an implemented composition method which incorporates elements of all three algorithmic styles. Then, techniques and motivations for building composite players out of several compositional agents are reviewed.

4.7.1 Solo

The *Solo* method is available as a level 1 module; however, the algorithm it implements is not a transformation of material presented at the input, as are

the other level 1 modules. The only relations *Solo*'s output bear to the events in the input block are that it is harmonically related, and its horizontal density in time will tend to increase and decrease along with the density of the input material. In that respect, *Solo*'s behavior is related to the transformative class of algorithms we have already extensively considered; it is, however, not strictly a transformation, since the input events themselves remain unchanged. Their only effect is to guide the generation of the module, through their harmonic content and horizontal density.

Solo is a hybrid of the transformative and generative styles: we have already seen the sense in which it is transformative. In fact, that part of the algorithm might more properly be termed simply responsive, since it looks at the input, but does not change it. The output of the module is a monophonic melody; the part of the algorithm that produces pitches generates them from an array of interval preferences, matched against reports coming from the chord agency. This part of *Solo*'s operation is more generative, the second algorithmic style included in the module's hybrid form.

Each time it is called, *Solo* adds between zero and four new events to each event in the input block. The offset between events will be chosen at random between twenty and fifty centiseconds. An array of intervals helps determine the pitches of the new events, through the following calculation: `next = (WhichChord(DUMP, 0)/2)+72+melody[rand()%5];` First, the root of the current harmonic area is determined by querying the chord agency. The answer returned is divided by two, to discount the mode of the harmony. Then, the root is added to 72, to place the activity of the *Solo* module at least two octaves above middle C. Finally, one of the intervals from the melody array is chosen at random, and added to the other elements. If the pitch calculated for any event is the same as the pitch of the event before, it is shifted down one half-step.

The offsets for all the new events are added together, and recorded. If the *Solo* module is called again before all of the events from the most recent invocation have been performed, the module returns without producing any new events. It is this part of the algorithm that tends to match the horizontal density of *Solo* to the density of the activity around it. Output from the

module will never overlap itself; all of the events from one invocation are guaranteed to finish before the next group begins. If the input events are quite sparse, some time will pass before a new event spurs fresh output from *Solo*. For inputs beyond a certain density level, however, *Solo*'s output will be more or less continuous; a new event triggering more response will present itself quite quickly after any given group has finished playing.

4.7.2 Composition Networks

Solo is a self-contained example of a hybrid algorithmic music generator. However, the concept underlying the facilities provided by *Cypher*'s composition section is one of combining and coordinating the actions of many small, independent agents. Rather than providing a set of stand-alone generators, many customized, responsive counterparts to *Solo* can be built using compositional agents embodying a variety of algorithmic styles.

Though there are many ways to combine methods, still some natural strategies arise. First of all, the transformation filters should be applied late. This is because both the generative and sequenced styles manufacture material from first principles, as it were: they do not require an input. Using transformations on the output of a sequencer or compositional algorithm will tailor that material to the situation at hand: particularly modules such as *Harmonize* or *TightenUp* can be used to adjust the output of other operations to match the harmonic and rhythmic activity being reported from the listener.

Second, even though sequences and algorithmic generators produce material from self-contained elements, as much guidance as possible to the generation method should be given when the method is first invoked. As we have seen, all of the compositional algorithms respond to performance parameters such as duration, pitch area, and so forth. Tuning these parameters as a function of the musical situation into which the result is to be introduced, will improve the fit of the new material to the existing context. Further adaptation can then be accomplished through modification of the generated material, using transformation filters.

Finally, the use of sequences can be adapted to the rest of the material flowing from the composition section by reading the sequenced events into an event block, and passing the block on through a listening section. There are two options here: either the sequenced material can be treated as if it were coming from the outside world, and processed by the main listener/player pair, or it can be treated as an output of the player, and edited by the critic. In either case, transformations could be used to coordinate the harmonic and rhythmic material of the sequence with a broader context, and pattern matching could be focused on the sequence to find significant patterns, which will then evoke their own response.

Chapter 5

Societal Architecture and Representations

The basic architecture of *Cypher*, on both the listening and playing sides, combines the action of many small, relatively simple agents. Agencies devoted to higher-level, more sophisticated tasks, are built up from collections of low-level agents handling smaller parts of the problem and communicating the progress of their work to the other agents involved. We have seen in the discussion of the listener, for example, that the process of chord identification is accomplished by an agency of interconnected feature classifiers and harmony experts.

This chapter will discuss in depth the design of an agency-based program architecture, describing the methods of communication and coordination that go into building up high-level processes from smaller agents, and the way a particular view of musical comprehension and composition was captured by such a design. First we will review in some detail the basic construction of the program, and the representations used for various tasks. Gradually, a picture showing the cooperation of many agents will emerge, enabling a look at the shared responsibility for various tasks spread through the program.

5.1. Listening Agencies

In describing the operation and communication of agents in *Cypher*, I will initially observe the usual separation of listener and player processes. Since the communication between these two components is a critical part of the overall design, such coordination will be introduced and examined shortly after a review of how the agents work within their primary area of expertise.

5.1.1 Feature Agents

When a *Cypher* event is sent to a listener, first of all a collection of low-level feature classification agents is trained on it. These agents can be considered as independent of one another, at least at this stage. Detailed descriptions of them are found in Chapter 3.

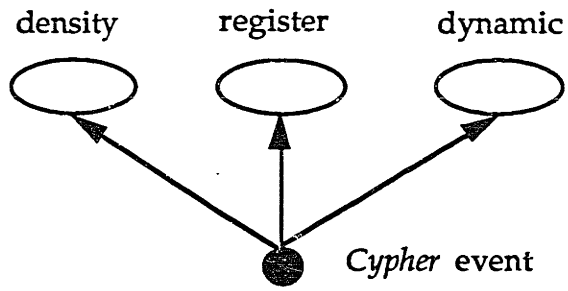


Fig. 32

As described in section 3.1.5, *Cypher* events are built from incoming MIDI data, and can represent either single notes or chords. In Figure 32, we see three feature agents analyzing a new *Cypher* event. (In this and all subsequent illustrations, the arrows indicate the direction of information flow. In Figure 32, data from the *Cypher* event is going up into the feature agents.) The three features shown (density, register, and dynamic), are all calculated at attack time. Only one of them is context-dependent, and therefore requires access to memory: the scale against which the register classification is made has been established by the analyses of preceding events.

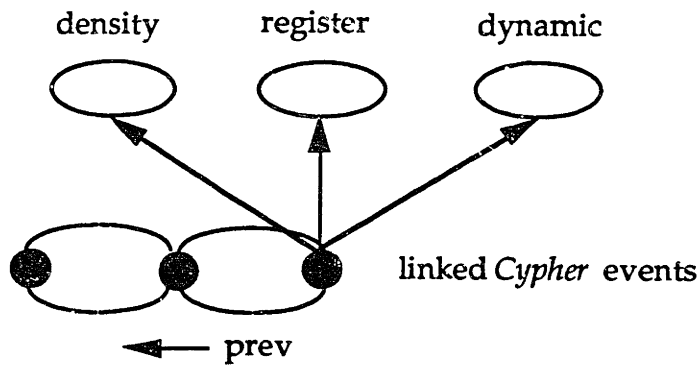


Fig. 33

In the expanded Figure 33 above, the context of previous events has been added. The register agent, then, classifies the new event, within the context

established. Other featural agents, moreover, depend directly on earlier events to perform their analysis task. In another expansion of the original processing diagram, shown below, we see the addition of two more feature agents: one measuring the speed of attacks, and the other tracking duration.

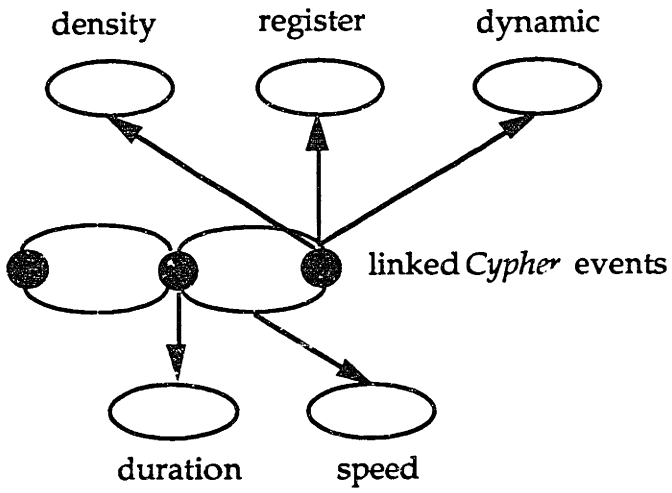


Fig. 34

The speed agent classifies the temporal distance between two events, and therefore relies on one event before the current one to make its calculation. Accordingly, the very first speed classification will be meaningless, since it only measures the length of time between the beginning of program execution and the performance of the first note. An even greater reliance on previous information is shown by the duration agent: because featural classification is done at attack time, it is impossible to analyze the eventual length of the new event. For that reason, the duration classification for the *previous* event is included in the current report.

Rather than a limitation, this procedure seems to reflect the way music is actually heard; the program will react to the duration of events closer to their release than to their attack. It can happen, however, that a new event will arrive before the previous one has been released. Each event is assigned a provisional duration at attack time; in the situation just mentioned, the provisional duration will be the one analyzed for the previous event. Normally, such a situation will not arise, but it is certainly possible, even to

be expected occasionally. For that reason, the provisional duration given to new events is the duration recorded for the last event released; even though this duration may correspond to an event played some significant amount of time previously, it will still have more resonance with the current context than would some arbitrary constant.

5.1.2 Chord Agency

The next step in the classification of an input event is the local harmonic analysis. Though the sequence of level 1 listening is here presented as a serial process, note that all of the analyses described so far could be done by the same collection of agents working in parallel. It is in fact only the operation of the host computer on which this work is implemented that keeps the feature agents working serially; an important design goal of the program is to make it amenable to future parallel implementations.

As shown in the illustration below, the chord agency accesses the new *Cypher* event directly, to find the values of the pitch information associated with it. At the same time, the other featural agents report their classifications of the event to the chord agent, guiding the evaluation of the raw pitch data. As we saw in section 3.2.2, the chord agency uses a connectionist network to find a plausible root for the incoming pitches. The network is assisted by the feature agents, whose reports will change the weights associated with certain pitches, according to their classifications. As an example, recall that pitches with a low register classification will tend to receive more weight than pitches higher in the range.

The formation of an agency incorporating a specialized, connectionist network, informed by the reports of other agents, gives the first real ordering to the processing of the level 1 listener. The low-level, perceptual feature agents (register, density, etc.) must complete their work before they can accurately inform the chord agency of their classifications. In a parallel implementation, then, a two-stage process would result: first, the feature agents arrive at their classifications in parallel; then, the chord agency accesses those classifications and the raw note data to find the dominant pitch of a local harmonic area.

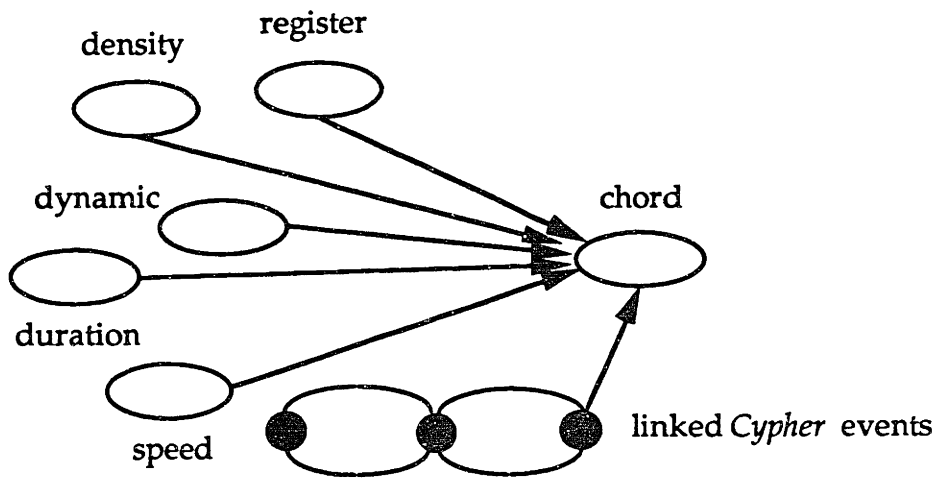


Fig. 35

5.1.3 Local Memories

The agency responsible for detecting beat periodicities shows an organization strongly paralleling that of the chord agency. Once again, a connectionist core is informed by featural agents, such that certain weights will be altered according to the classifications reported. Both the beat agency and the chord agency maintain separate, private memories, which record the activation levels of various theories. Further, there may be several copies of the listener active at the same time, and each copy will maintain separate memories for its constituent agencies.

For every distinct stream, the beat and chord agencies communicate results with each other. In the figure below, we see the beat and chord agencies, each coupled with their own local memories, and receiving information from the low-level feature agents. Again, this organization reinforces the pattern of two-stage processing on level 1 that we noted earlier: first the perceptual feature agents (register, density, etc.) arrive at independent classifications. Information from these features, from the raw event data, and from the other second-stage agency, are processed by the chord and beat agencies after the perceptual agents have finished, making use of contexts recorded in their local memories.

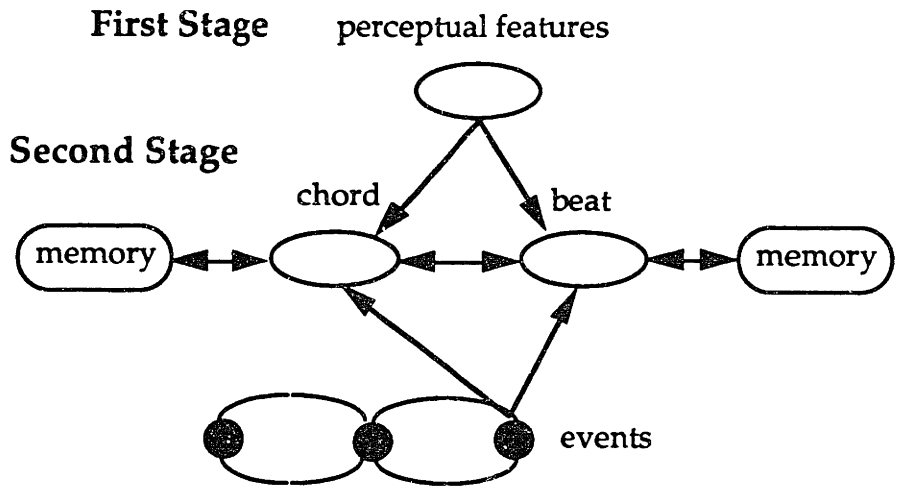


Fig. 36

5.1.4 Mutual Reinforcement

In Figure 36, we see the first two-way connections, signifying communication paths where each end can be the source or destination of information travelling across the connection. Obviously such communication holds between an agency and its memory; the presence of two-way communication between agencies, as we note between the chord and beat agencies, is a more problematic issue.

When two agencies are cooperating, making their calculations interdependent, a situation arises in which the ordering of execution and communication between the two sides becomes unclear. In the case of the beat and chord agencies, certain types of results arrived at by either agency can further some interpretation calculated by the other. If the beat agency reports that a new event was on the beat, the pitch information associated with that event is given more weight by the chord agency. Conversely, if the chord agency finds evidence of a new local harmonic area, that finding tends to influence the beat agency to find a beat boundary. The problem then becomes one of deciding which agency should have a stronger influence on the other, or what message should be passed in which direction first.

In this case, a plausible way to arbitrate the communication between the chord and beat agencies is to use the confidence levels each of them maintains: messages with a high confidence rating are given more weight by their recipient than low-rated messages. Still, the question of execution order remains. Because the inter-agency influence is mutual, there is no clear way to decide whether partial results should be communicated from the chord agency to the beat agency first, or vice versa.

In a single processor implementation, like this one, the question must be decided: in *Cypher*, the beat tracker sends a message to the chord identifier first, followed by a message in the opposite direction. I regard a beat boundary as stronger evidence for the significance of the associated harmonic information, than I do the identification of a new chord as evidence for a beat boundary. Further, the beat tracker is making predictions about the arrival of the next beat. The chord agency can therefore plausibly influence the belief in the beat prediction after an initial estimate has been made. The processing chain proceeds as follows: the beat tracker uses the event and feature data to make an initial prediction, and to classify the current event as on or off the beat. Second, the chord agency decides on a classification, referencing the beat tracker's *onbeat* decision. Third, a change in chord will send a message to the beat tracker, altering the confidence in the current theory, according to whether or not the chord change coincided with a beat boundary.

5.1.5 Attachment of Classifications

At this point, the raw *Cypher* events processed so far are assigned a featurespace classification, which records values for the perceptual features and local harmony. In figure 37, the featurespace word is shown as an added layer of information surrounding the original event. Attaching the classification to the event allows us to accomplish two objectives: detailed information about the nature of the event is made available to level 2 processes, which observe the way such information changes over time; and, the judgement of all individual agents concerning an event can be ascertained by referencing it, to further the processing of any agency which might require the information.

Basically, there are two ways to implement the connectivity of agents into larger agencies: either the results of the agent can be broadcast through messages to all concerned agencies, or the results can be attached to the event being analyzed, where it is available to any interested party. The first method is advantageous for parallel architectures, when the results can be sent out to many processes working simultaneously; the second is more efficient for serial implementations, where the desired result can be simply read off of the event, rather than obtained through a message from an agent.

Moreover, the commitment arising from either method is slight; replacing the attachment procedure with a mechanism for sending results out to a list of subscribed processes, or in response to a query, is not a major undertaking. This implementation uses the attachment approach.

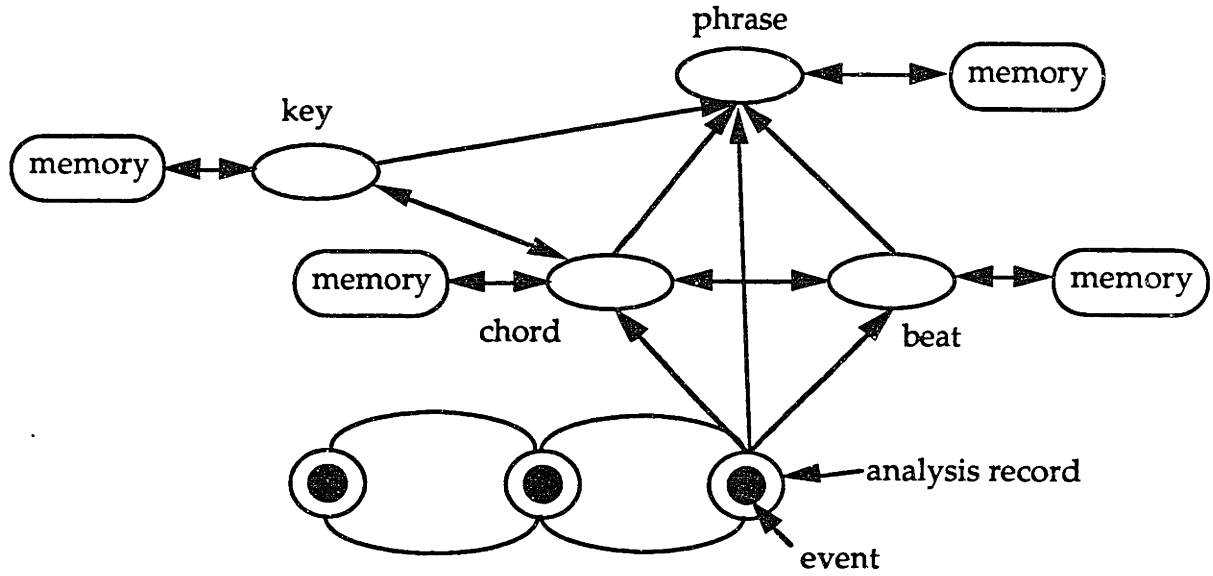


Fig. 37

In figure 37, notice the addition of two agencies, one which tracks key (the phrase-level harmonic analysis) and another which looks for phrase boundaries. Again, these agencies maintain local memories, which are also kept distinct per event stream. In other words, the key agency has access to a local memory, which communicates with no other process. Further, there are separate local memories for the key agent associated with all active event

streams. In *Cypher*, then, there will be two local memories for the key agency, because there are two active listeners: one corresponding to the stream of events coming from the outside world, and another associated with the event stream produced by the composition section. The key agency is involved in a mutual communication link with the chord agency; the phrase finder communicates with chord, key, and beat, as well as referencing the featurespace classification for the current event – thereby effectively communicating with the feature classification agents as well.

5.1.6 Analysis Record

We have seen that a featurespace word is attached to each *Cypher* event, retaining the classifications of the initial stages of listener analysis. The later stages, typified by the additional agencies seen in figure 37, attach the results of their calculations to the event as well – extending the layer of information which can be accessed by other agencies. Again, this process of attachment is an implementational choice enabling the communication of the results of one agent to all other agencies requiring that information.

Even if communication were effected through message passing to a list of subscribers, we would still need to attach much of this information to the event – because the abstractions represented in the attached data are used to generate the higher level descriptions of the evolution in event behavior over time. In the next section we will see how types of change are processed; first, let us examine the representation of analysis information attached to each *Cypher* event.

```
typedef struct ar {
      struct ar *prev, *next;
      char phrase, onbeat;
      short key, chord, function, score, chord_certainty;
      unsigned short featurespace, same, regular;
      } Analysis_record;
```

The complete attached layer of information is stored in an *Analysis_record*, shown above. Analysis records are, like the underlying events, doubly linked lists, preserving the progressive perspective on the events they represent. The

prev and *next* pointers lead to analysis records associated with events earlier and later in time, respectively. *Phrase* and *onbeat* are flags: *phrase* signals event which are on phrase boundaries. Boundaries are marked by events beginning the new phrase; *phrase* will be true for the first event of any phrase. The *onbeat* flag is true for events which are on the beat, according to the beat agency.

The next collection of fields refer to classifications arrived at by various listener agencies. *Key* and *chord* keep the results of those two harmonic evaluations. The *function* is calculated from the key and chord entries, and indicates the function of the chord in the key: this supplements chord identifications of a given root with the scale degree of the root relative to the key, denoting traditional tonic relations such as tonic, dominant, subdominant, etc. The *score* is the numerical evidence for a phrase boundary associated with this event. If *score* exceeds the boundary threshold, maintained elsewhere, *phrase* will become true, and a new phrase begun. The *chord_certainty* is the strength of activation of the dominant chord theory relative to the activation spread through all possible theories, and gives a measure of how strong the leading theory is relative to the rest.

The *featurespace* word is the collected reports from the feature classification agents. The following two fields, *same* and *regular*, are running statistics kept by the level 2 listener. *Same* indicates which feature reports have stayed the same between the current event and the previous one. *Regular* is the result of the level 2 analysis – bit fields in this word indicate, for each feature, whether it has been behaving regularly or irregularly over the current phrase.

5.1.7 Level 2

The listener on level 2 characterizes the way lower level classifications change over time. The analysis record attached to each *Cypher* event is used to make this calculation, and the results are added to it as well. In figure 38, notice that the regularity detection agent has been added to the picture of the listener we have been constructing in the course of this section.

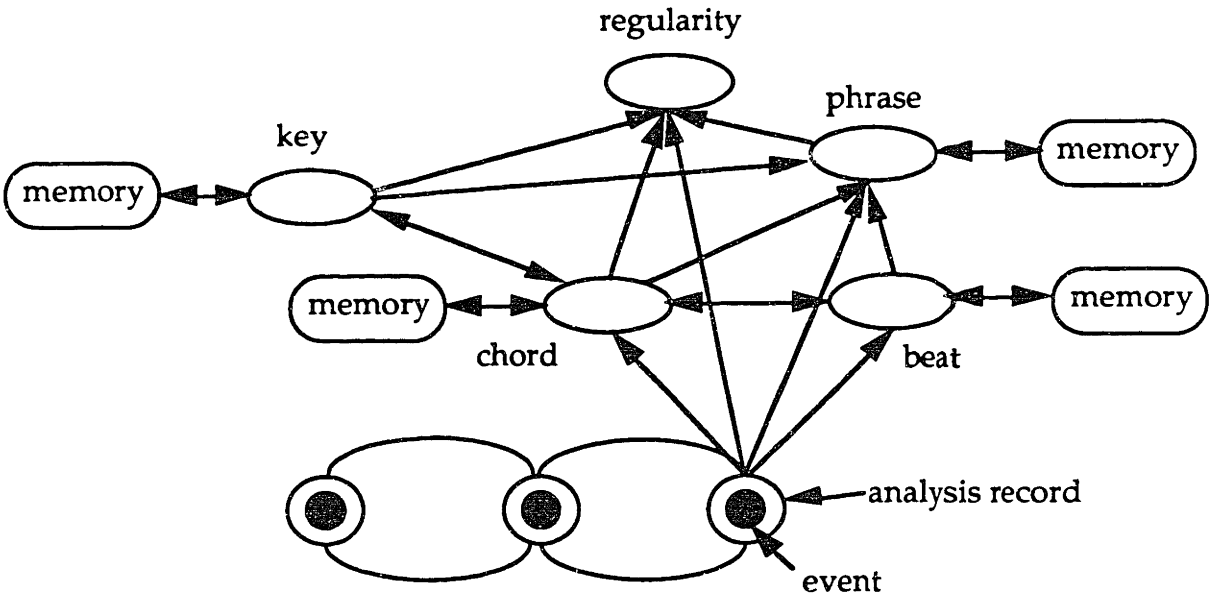


Fig. 38

The regularity agent refers to the featurespace word, the difference between the current featurespace and the previous one, the running statistics of classification change during the current phrase, and the chord, key, and phrase agencies. A detailed description of how this analysis works can be found in Chapter 3: the point to be noticed here is the connectivity and quasi-hierarchical arrangement of the various processes involved in a listener. The important characteristics of this architecture have been reviewed in turn during this section: preservation of the progressive perspective – that is, continual access to information associated with events proximate in time; maintenance of local memories on a per-stream basis for each agency; mutual reinforcement of cooperating agencies; the attachment of analytic results to the events classified, where they can be read by other interested processes.

The illustration in figure 38 provides a reasonable picture of the architecture of *Cypher*'s listener. However, it is at a significant level of abstraction; as we have built up this picture, lower level details (such as the feature agents), have been boiled down to the analysis record attached to the original events. With the understanding that the figure is a useful reduction of the entire collection of calculations carried out, we see the real outline of the type of organization I referred to in Chapter 2 as a "layered network". Recall the

claim that the listener embodied a hierarchical analysis of an event stream, but that the hierarchy was tree-like in only a very restricted sense.

Here we see the structure behind the claim: the organization can be considered hierarchical, in the sense that the regularity description subsumes individual events in a characterization of a group of events. At the same time, the progressive and communication links between events and the processes analyzing them renders the architecture as a whole more like a network of interconnected agents. And, this is only half of the story. Next we will look at composition networks, and finally sketch the structure encompassing both competences.

5.2 Composition Agencies

In Chapter 4, we examined the building blocks of compositional networks, as these are constructed in *Cypher*. In this section, we will consider in more depth the nature of the networks linking these blocks together. Recall that there are three strategies available for generating music from the composition section: playing sequences, generating new material, or transforming received material. In contrast to the parallelism of the listener, compositional agencies built from these elements tend to link many processes together in chains, with the music being generated passed along from one link in the chain to the next.

5.2.1 Serial Processing

The way several transformations of some material are invoked is the most obvious manifestation of the chain-like behavior of many *Cypher* composition networks. When several transformations are due to be applied to a block of events, these are executed in series, with the output of one transformation passed along to the input of the next.

transformations

Fig. 39

It is remarkable that the processing networks on the composition side are organized so differently from those of the listening side. Both competences use many small agents to build up more complex behaviors, but on the listening side these can operate largely in parallel, while on the composition side they function largely in series. Again, it is largely a reliance on the idea of transformations that engenders such a structure. Applying a set of transformations to some base material in parallel is, first of all, not possible on the current hardware platform, and, second, would produce highly variable results from event to event. We have already seen that the order of transformations can make a big difference in the eventual result; transformations all operating on the same material in parallel would have no fixed order, and the sounding result at the output would show wide deviations for even identical input material.

With the model of composition implemented in *Cypher*, there really is no way around a significant amount of serial processing. However, the real interest of the composition sections lies in the way decisions are made to apply some transformations and not others, and in the way some material is chosen as the basis for transformation. The way these decisions are expressed, evaluated, and altered, is where the parallelism of the architecture as a whole again comes into play. In the next section, I will explore the critical question of how the operation of one or more listeners is coordinated with that of the composer.

5.3 Cross-Competence Communication

The composition section is a collection of methods. Some of them generate material, and others perform transformations. As we have seen, they are generally applied in series. The methods chosen, and the operation of the methods themselves, are a function of messages arriving from some *Cypher*

listener. In the current implementation, there are two active listeners: one analyzing input from the outside world, and the other performing the function of an internal critic, reviewing and altering the output of the composition section before it is sent on to the outside world.

5.3.1 Execution Conditions

A *Cypher* listener performs several levels of analysis, and maintains a record of the classifications calculated for many different musical features and their behavior over time. The record containing this information is sent on to a process which organizes the invocation of composition methods, making the output of all listener components available for compositional decisions. According to the response connections established between incoming listener messages and composition methods, anywhere from zero to a dozen or more methods will be called in series, in the order of their priority.

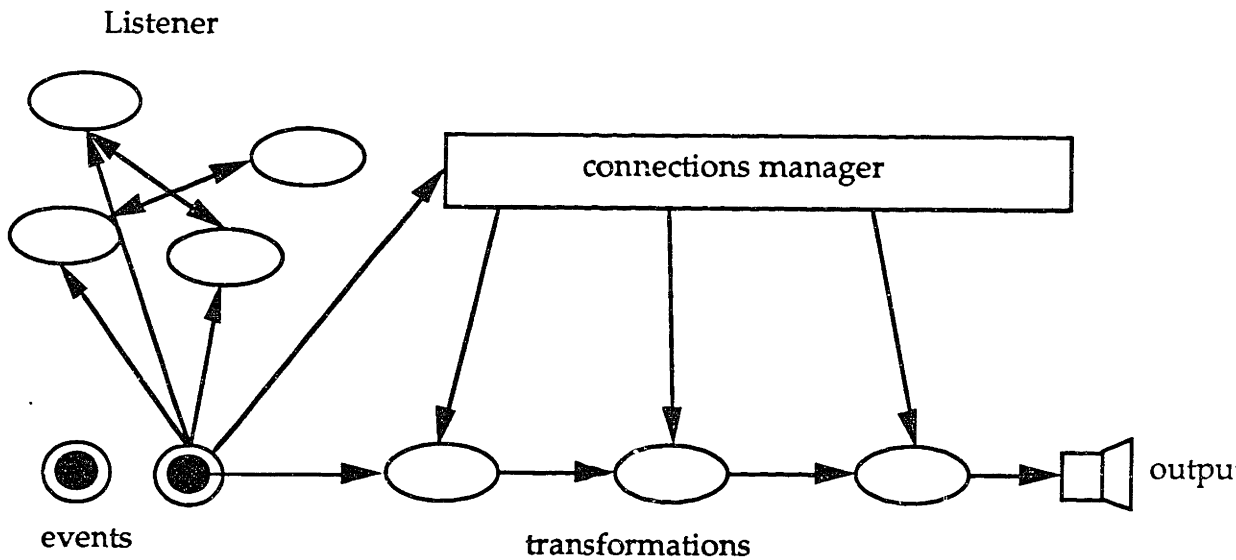


Fig. 40

In Figure 40, we see a stylized version of the listener representation built up in the previous sections, on the left. At the bottom of the listener, incoming events, with their attached analysis records, are shown. The *connections manager* box is receiving the analysis record for the current event. According to the connections maintained there, certain transformations will be called

up, and ordered by priority. These are then applied to the same input event classified by the analysis record, and the result of all transformations sent on to the output. In this example, the seed material for transformations is the input itself; another function called by the connections manager might decide to choose other material as a base, in response to some listener message.

The connections between listener messages and composition methods are established as logical expressions on the features and regularities reported by the listener. Consider the following example: High OR Loud -> Trill. Here, the Triller transformation is queued up to be applied to some seed material, if the analysis record associated with the current event shows the event to be either high or loud. The job of the connections manager, then, is to parse and evaluate the logical expressions on the left-hand side of such a connection, and prepare for execution the appropriate composition methods found on the right-hand side.

The connections as represented on the graphic interface allow only inclusive OR as a logical operator. Attaching two features to some method means that when one or the other feature is found, the method will be readied for execution. Any number of features can be tied to responses in this way; however, they are all related to each other by this OR relation. The interface is not, however, the only way to specify connections. The scripts of response connections executed during score orientation allow the connections manager to be programmed using a slightly wider range of logical operators. Through the script mechanism, inclusive OR and NOT operators are available for connecting any feature or regularity report to a method. For each feature, the connection manager searches two lists in the Link structure: if the feature is asserted in the analysis record for the event under consideration, all methods in the OR list will be queued for execution. If the feature is negated in the analysis record, all methods in the NOT list are readied.

The full range of logical expressions should be supported by the connections manager. The path to completing it is clear; what is needed is a fully developed parser, a compact representation of the notated expressions, and extensions to the connection manager which, for each incoming event, would evaluate the appropriate parts of the analysis record to see if any active

expressions are satisfied. Further, full nesting of expressions should be handled by the same mechanism, so that compound statements such as (Soft OR High) AND NOT Fast can be specified as well.

5.3.2 Aesthetic Productions

Another component in the collection of processes establishing interaction between listening and composing has been added to Figure 41. The listener/connections manager interaction described in the previous section is shown at the left of the figure. Now, we see two more stages added between the transformations applied by the connections manager, and the output of data to the synthesizers: first, a second copy of the listener analyzes the output emanating from the collection of methods called up by the connections manager. This second copy of the listener is labelled a "critic" in the figure.

Actually, the second listener is only half of the critic. The other half is made up of the rules shown in the box marked "productions." As we saw in Chapter 4, the critic includes a number of production rules, which examine the output of the composition section, and apply further modifications to it when conditions specified in the productions are satisfied.

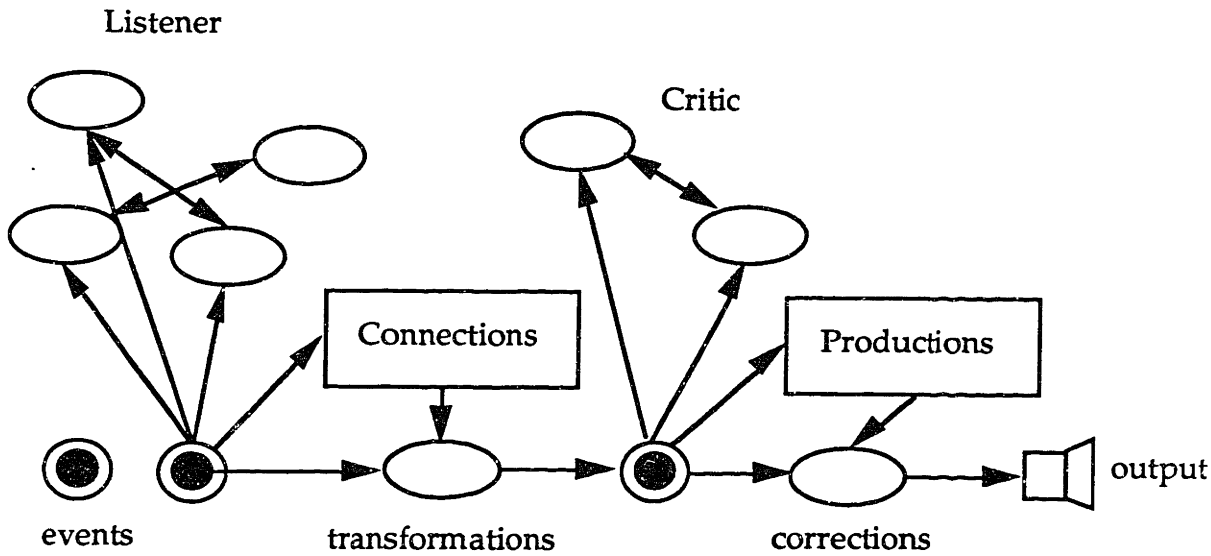


Fig. 41

There are two major differences between the operation of the connections manager and the critic's productions. First, the compositional methods invoked by the connections manager in response to listener messages are specified by the user, either in a script of connections, which is stepped through by the score orientation mechanism (Chapter 7), or by using the graphic interface, which allows the user to draw connections between messages and methods on the screen. The critic, in contrast, maintains a number of rules associating responses with listener messages under certain conditions. These connections, then, are applied automatically according to the musical context. Often, the interface will be used to try out several analysis/response combinations, the most useful of which is saved to the productions for automatic application in performance.

The second main difference is that the critic's connection rules can use the full range of logical expressions on listener messages. We saw in the previous section that the logical operators available to the connection manager are currently limited to inclusive OR and NOT. In the critic, any arbitrary expression can be written as the conditional part of a production rule. A set of classification-reading routines can get the analysis results from the information attached to any event. Using these routines, conditional statements such as

```
if ((LoudVal(featurespace)>0) && (RegVal(featurespace)<3))
```

can be written to any level of complexity, taking advantage of the normal C language evaluation mechanisms. The example expression means that the action associated with this condition should be executed if the output of the composition section is louder than the quietest possible value, and not in the highest register. Accordingly, critic production rules can be sensitive to more complicated conditions and combinations of input features than can the connections manager links.

5.3 Established Approaches

Several applications have addressed problems of knowledge representation, inference, and learning in music. In this section, I will review some of these applications and the particular representational or inference techniques they employ. For each in turn, equivalent structures in *Cypher* will be considered.

5.3.1 Production Systems

Rule-based, or production, systems rely on a collection of rules with the form: if <condition> then <action>. In choosing an action to perform, a representation of the current situation will be examined to see if features mentioned in the condition part of the rule are in the state required by the condition. If so, the action part of the rule fires and makes some change to the state.

A musical application of this approach is Stephan Schwanauer's *Music Understanding System Evolver* [Schwanauer88]. MUSE uses a generate-and-test method to perform tasks like supplying a soprano voice to an unfigured bass. The knowledge encoded in the production rules accepts or rejects generated solutions and directs further trials using some understanding of voice leading. These further trials are a form of *backtracking*, which occurs when a test has indicated failure. Then the system backs up to an earlier state and picks up processing there with a different solution from the one which led to the failure.

Rule-based systems have found some use in interactive systems; a notable example is the *Interactor* program designed by Morton Subotnick and Mark Coniglio. In *Interactor*, scenes are constructed which consist of a set of production rules. The conditions of each rule are matched against the state of MIDI input and the program's internal state : the first rule whose condition matches is executed. One possible action for a rule is to jump to another scene, activating a new set of conditions and actions. The similarity to production systems is limited to the form and processing of the rules; there is no generate-and-test cycle, nor any backtracking.

In this limited sense of productions, *Cypher* can be regarded as another example. The messages from the listener to the player are sent as a function of features found in the environment; certain player processes are invoked when messages corresponding to particular feature configurations are sent. This again is a form of production rule: the <if> part is made up of listener states, and <then> corresponds to the player processes called by the listener.

Generate-and-test or backtracking mechanisms are again absent. *Cypher* is in one respect even further removed from the realm of production systems than is *Interactor*: the <then> part of a *Cypher* production, the player processes invoked, have no effect on the state examined by the <if> condition. The only exception to this observation arises when the program is running introspectively: then the compositional operations invoked by the action part of the rule produce events which are fed back into the listener, effecting changes in the state examined by the production conditions. As we have seen, two sets of productions are maintained in Cypher. The first set is manipulable directly by the user, either through the interface or through a score orientation script. The second set makes up the program's internal critic, which reviews output about to be played and alters it according to the production rules representing its aesthetic preferences.

5.3.2 Frames

Another representation which has been extensively developed by the artificial intelligence community is called a *frame* [Minsky86]. A frame is a collection of related information, representing typical features of some situation, which will be used to direct processing whenever a variant of that situation is encountered. The frame consists of a number of *terminals*, which correspond to usual features found in the frame's reference situation. For example, a frame for a chair might have terminals representing legs, a seat, a back, etc.

> Much of the phenomenological power of the theory hinges on the inclusion of expectations and other kinds of presumptions. *A frame's terminals are normally already filled with 'default' assignments.* Thus a frame may contain a great many details whose supposition is not specifically warranted by the situation. [Minsky75 pp. 246-247]

The level of description captured by a frame corresponds well to the higher levels of a music listening task. A machine listener should be able to recognize common situations, and note where the details of what is really happening deviate from what might be expected. The organization of information in frames, including default values or processes for certain slots, indicates that they are a good representation for such kinds of analysis.

A frame matcher could observe features of the current context and compare them to a library of known frames. The frames would include behavioral and pattern information. Once sufficient behavioral or patterned aspects of the incoming context corresponded to a known frame, that frame would become active. The default assignments in the frame would then represent common expectations concerning the context found; according to the terminals of the frame, the listener would be directed to look for specific confirmations or contradictions of feature behaviors, or expected continuations of harmonic, melodic, or rhythmic patterns.

The structures implemented in *Cypher* are directed toward the eventual realization of a frame-based music understanding system. Frames built using *Cypher* elements would have terminals corresponding to the listener messages now supported. Therefore, slots for feature classifications, beat period, harmonic activity, and the change of these through time would be collected and stored as the characteristic parts of some musical context. Such frames would, I believe, bring us closer to the situation outlined in the preceding paragraph. The obstacles remaining have primarily to do with the limited number of analysis levels. Now, characterization extends to the phrase level. Frames for typical phrases, however, would be less useful than higher-level structures capturing the behavior of longer, more characteristic styles. As the classifications produced by *Cypher* begin to expand upward, gaining in descriptive scope, frames gathering many listener messages together will provide a more meaningful representation of musical behavior.

5.3.3 Scripts

The passage of time is a critical concept to be included in any computational model of musical activity. We need to be able to manipulate operations on many temporal levels, and indicate their starting time, duration, rate of speed, etc. A *script* is a representation which defines an ordering for events. A restaurant script, for example, describes the typical situation of entering the restaurant, examining the menu, ordering, eating, paying, and leaving [Schank&Abelson77].

For a moment, let us consider the generation of a musical phrase, with the realization that similar questions will arise on other temporal levels, such as groups or parts of phrases. If we can specify the kind of motion traversed by the phrase, in terms of its harmonic, textural, and rhythmic functions, we can be directed to operations appropriate to it. A cadential phrase could include a pointer to a time map which might shape the rhythmic events to effect a slight ritardando leading to the cadential point. Again we will want to have representations of typical situations, such as in the preceding example.

We can combine the idea of a script with that of the frame in a representation known as a *trans-frame*. A trans-frame is a frame including certain kinds of agents: actor, origin, trajectory, destination, and vehicle are typical trans-frame terminals. Some questions a trans-frame will be used to address are : "Where does the action start? Where does it end? What instrument is used? What is its purpose or goal? What are its effects? What difference will it make?" [Minsky86 p. 219] Trans-frames represent some kind of transfer, or transformation. They are useful on higher levels of both listening and composing tasks, and for the same reasons: to the extent that a musical discourse shows a goal-oriented behavior, or makes recognizable variants of some pre-existing material, such behavior can be represented, at least in part, by a trans-frame representation.

In the previous section I introduced the concept of *Cypher* frames. Extending these to the trans-frame model would require important new kinds of information. In particular, trans-frames will often note the reason for some activity: their suitability to goal-directed behaviors is to large degree a function of their representation of the motion's purpose. In their simplest form, frames simply associate features with values. The analyses performed by *Cypher* could already realize such a structure, particularly after extensions to higher levels, as noted above. Constructing *Cypher*-like trans-frames, however, would require much more additional functionality. Rather than considering simply what is typical, or which things usually go together, some notation of a gesture's musical purpose would be needed to realize the full power of the representation.

Chapter 6

Pattern Processing

Pattern processing encompasses two goals: 1) learning to recognize important sequential structures from repeated exposure to musical examples, and 2) matching new input against these learned structures. I will refer to processing directed toward the first goal as *pattern induction*; processing directed toward the second will be called *pattern matching*. Used in this sense, pattern induction is an algorithm for finding structure in a stream of live performance data. We want to find sequences, strings of number representing such things as harmonic progressions, rhythmic patterns, or melodic fragments, which gain significance either through repeated use, or because of their relation to other known structures. Pattern matching, in contrast, is an algorithm for comparing new input to the known sequences found from pattern induction, and signalling new instances of the stored patterns.

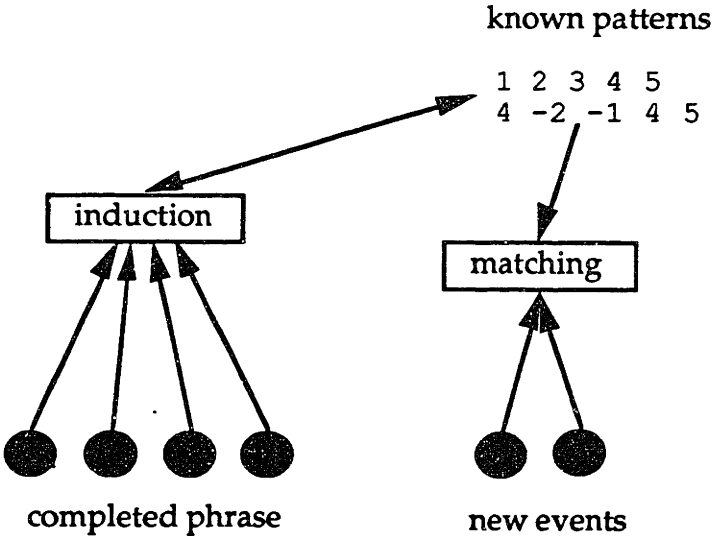


Fig. 42

The two kinds of processing have different effects. A successful application of pattern induction will yield a new pattern, to be added to a database, which is retained in the program's long-term memory from execution to execution. The pattern matching process can send out several different messages, depending on its progress through a match. For example, the matcher will

emit one message at the completion of a successful match, identifying which pattern was found; another kind of message will signal an ongoing match, when the beginning of some pattern has been seen.

An assertion that the beginning of a known pattern has been found allows the composition section to schedule events out of the continuation of that pattern. If the remainder of the sequence continues to follow the known structure, the player can anticipate future events and schedule output to coincide with it. Discovered patterns are held in a long-term memory, which is saved and restored with each run of the program. Long-term memory retention provides the program with an expanding repertoire of known melodic and harmonic sequences, which can be matched against in the listening process, or used as raw material in compositional methods.

6.1 Precedents

Precedents to this kind of pattern processing are found in several sources, musical as well as non-musical. Pattern recognition, a field which would seem to bear a strong resemblance to pattern induction, is in fact quite distinct; pattern recognition's most common area of inquiry is computer vision, and encompasses several techniques developed for finding objects in a raw image. The patterns dealt with in this thesis differ in two important respects from vision problems: first, the patterns we seek to find are played out over time; and second, our patterns gain significance through repetition and their resemblance to known structures, rather than through the intrinsic qualities of any individual instance.

Pattern matching is a technique which has received attention in a number of fields; the specific form of the problem addressed here again assumes an aspect somewhat different from the norm because of the musical application involved. Strict matching of precisely defined patterns is only a beginning for the processing needed to deal with musical sequences. Deviations from an exact repetition of stored patterns can arise from many causes, and a musical pattern matcher needs to be able to accommodate these deviations: examples include transposition, change in ordering, omissions, and variations.

Several researchers in computer music have worked in this area, with contributions ranging from the pattern induction techniques of David Cope, through the pattern description notation developed by Simon and Sumner, to the sophisticated matching algorithms pioneered by researchers such as Roger Dannenberg and Barry Vercoe.

6.1.1 Cope

In our parlance, pattern induction is the assertion that certain sequences of numbers have been repeated to such a degree that they should become marked as significant, and remembered for later use. David Cope's *Experiments in Music Intelligence* (EMI) are an example of pattern induction in this sense: large samples of music from particular composers are analyzed by the program to extract patterns, or "signatures", which exceed frequency thresholds maintained by the software.

> This article assumes that 1) pattern matching is a powerful technique to use in attempting to discover some of the reasons why composer's styles sound as they do; 2) patterns judged as alike which occur in different works of a composer are valuable and constitute what the author will call signatures; 3) re-using such signatures in music composed following standard chord protocols and voice leading can lead to replications in the style of the composer. [Cope90 p. 288]

In Cope's view, culling patterns from large collections of data, representing several examples of some composer's work, captures significant features of that composer's style. Demonstrations of the learned styles are built by pressing signatures into a prepared substrate of tonal material. For example, an EMI composition in the style of Mozart begins with a template of textbook sonata-allegro form, and adds signature patterns as motives and cadences to provide the proper Mozartian stylistic traits.

What I call induction Cope refers to as matching: the heart of his system is a group of routines which exhaustively search number sequences to find repeated series. The number sequences processed represent chains of intervals. The first step in the induction process reduces strings of pitches to strings of the intervals separating those pitches. In this way, the same

material beginning on different scale degrees will have an identical, intervallic, representation. Cope's algorithm then performs an exhaustive search of melodic sequences to find those which are considered identical, within certain tolerances.

One tolerance he uses is a *range tolerance,* which will allow intervals falling within the specified range to match. During the exhaustive search part of the algorithm, each successive interval in the two strings being matched is compared. Identical intervals will, obviously, be considered a match; intervals whose absolute difference does not exceed the range tolerance will match as well. For example, setting the range tolerance to 1 will produce matches between major and minor mode versions of a melody, because the scale degrees of the major and minor modes usually differ by no more than one half-step.

Another tolerance, or "tuner" as Cope calls them, is the *error tolerance,* which is the maximum number of times interval matches can fall outside the bounds of the range tolerance without aborting the match as a whole. If the error tolerance is non-zero, patterns with some small number of significant deviations in intervallic structure will still be recorded as patterns. This tuner allows variants of some sequence to be recognized as similar, where the variations go beyond the simple major/minor discrepancies handled by the range tolerance.

David Cope's EMI experiments are interesting, I feel, because they implement a serious method for finding significant sequences, where significance is regarded as a function of frequent repetition. This evaluation is somewhat different from Cope's own – he considers the program to be finding fundamental elements of a composer's style. That claim I find overblown: style is a combination of processes operating on many levels of harmonic, rhythmic, and formal structure. Cope's signatures correspond, in my view, to a library of a composer's clichés, rather than to a substrate of stylistic elements.

6.1.2 Simon and Sumner

In their 1968 article "Pattern in Music", Herbert Simon and Richard Sumner outlined a method of describing alphabets and their manipulation which represents significant features of temporal sequences. Moreover, Simon & Sumner consider patterns to be among the fundamental structures of music cognition, and in particular, to be a strong determinant of the expectations and goal-directed motion which are such a basic part of the experience of Western music.

> We are led by these studies to conclude that pattern-seeking is a common activity of people faced with temporal sequences, and that the vocabulary, or stock of basic concepts they have available for describing patterns is parsimonious, involving only a few elementary notions together with rules for combining them, and also is relatively independent of the specific stimulus material. [Simon&Sumner68 p. 222]

Simon and Sumner's agenda assumed that patterns are the primary building blocks of music – if an exacting notation of patterns and transformations could be developed, entire pieces of music could be described and generated from the basic pattern structure derived from some particular piece. Though I regard pattern processing as one perspective on musical experience, which only gains coherence in conjunction with other modes, I am nonetheless struck by the terms they employed to describe patterns and their function in music:

> Patterns involve *periodicity* – repetition (in a generalized sense) at intervals that occur periodically (in a generalized sense). Patterns make use of *alphabets* – sets of symbols ordered in a definite sequence. Patterns can be *compound* – made up of subpatterns which can themselves be represented as arrangements of symbols. Patterns generally possess *phrase structure*, which may be explicitly indicated by various forms of punctuation. Patterns, as we have already seen, may be *multidimensional*. Repetition in pattern generally involves *variation*. [Simon&Sumner68 p. 228]

Their treatment of patterns comprised two stages of processing, which bear some resemblance to my own: the first was called *pattern induction*, finding structure in a sequence of elements (I have adopted their terminology for an analogous operation). The second process was termed *sequence extrapolation*, the generation of an extension to some pattern according to the structure induced found through pattern induction.

"The pattern inductor may be thought of as a 'listener,' since it accepts the music (or the score) as input, and detects the melodic, harmonic, and rhythmic relations implicit in it." [Simon&Sumner68 p. 244] The work reported in this article is an intriguing approach to pattern processing, which relates the cognitive machinery of music perception to such tasks as letter sequence completion. However, I think the "magic bullet" status accorded it by Simon and Sumner was too optimistic: the predictive power they describe accounts for only a part of the full experience of music. Their technique, in particular, does not learn or retain any sequences, regardless of their importance or frequency in the examples it has seen. Moreover, it is not clear how their rules should be applied in real time; the rules for finding phrase boundaries, for example, require much more backtracking than is possible for a system trying to find boundaries as they occur.

My work, as we shall see presently, tends to emphasize the script-like qualities of patterns, giving weight to particular sequences according to their frequency of appearance, and searching for variations of those sequences in later examples. Simon & Sumner's theory forms what I believe to be a useful extension of the approach I have implemented, adding a more rule-based component to the basic pattern matching facilities I employ.

6.1.3 Dannenberg and Bloch

In their article "Real-Time Computer Accompaniment of Keyboard Performances", Joshua Bloch and Roger Dannenberg describe techniques for matching monophonic and polyphonic performances against stored scores representing the expected performance. For the initial stages of our pattern processing task, we need match only monophonic sequences, and have adopted the technique proposed by Dannenberg and Bloch to accomplish it.

The patterns *Cypher* currently considers are strings of numbers, which may be interpreted in various ways: as chord progressions, for example, or progressions of rhythmic durations. The patterns considered in [Bloch&Dannenberg85] are the Note On messages arriving from the performance of some musical score. Therefore, our problem corresponds to their monophonic matching case: we do not try to coordinate the reports of multiple listeners, or consider simultaneous harmonic or rhythmic streams, and so are able to avoid the polyphonic matching problem they describe later.

Briefly, the Bloch & Dannenberg algorithm computes a matrix relating score events (the stored representation) to performance events (the notes coming from the live player). Each time a new performance event arrives, the next column of the matrix is computed: finding the maximum rating of possible score to performance correlations will point to the most likely position in the score of the current performance.

This approach maps quite easily onto our problem: given an induced pattern, we want to match incoming information against it to see if the pattern is being followed. The induced pattern, then, corresponds to the score in the Bloch and Dannenberg application, and new input corresponds to their performance. There are, however, significant differences as well. First of all, pattern processing in *Cypher* requires matching against many different patterns simultaneously. Second, there is no a priori way to know when a pattern may begin; new input must be continually matched against the head of many patterns to find incipient pattern instances. The way these additional problems are addressed, and the specifics of the implementation adapted from Dannenberg & Bloch, will be addressed later in this chapter.

6.2 Predicting Musical Goals

Recognition of directed musical motion can take place in two ways, performed independently or in combination: either direction can be detected in the event-to-event behavior of the input, or the input can be recognized as following a stored template of typical musical patterns. The first kind of recognition is the one addressed by the work of Simon & Sumner: they

assume certain fundamental alphabets, such as scales, and a small number of way of organizing the elements of those alphabets. Building on these assumptions, they write processes to find patterns arising from manipulating the alphabets with the given operations. Once such a pattern has been found, they can extrapolate (predict) the extension of the sequence, by continuing to apply the discovered operations on the underlying alphabet.

As an example of the second phenomenon, imagine the fundamental tonal direction of a dominant-tonic cadence. To cooperate with the cadence motion, the system must know what the current key is, that the current chord is the dominant of that key, that the tonic of a key often follows the dominant, and something about the beat or rhythmic regularities of the context. Most of these tasks we have already described as parts of the *Cypher* listener: from the reports the listener sends out, we can find the key, the function of the current chord, and when the next beat is scheduled to arrive. Stated in the simplest way, the object of pattern processing is to give the listener the other piece of information needed: the knowledge that the tonic of a key often follows the dominant.

All of these tasks, carried out in parallel by lower level agents, must be coordinated by a cadence detection agency that decides on the basis of such information that a return to the tonic is imminent. Such a decision then communicates a goal to the generation section, which will similarly coordinate its composing agents to move toward the expected goal, for example by executing a decelerando, or crescendo, or by performing voice-leading operations moving toward the tonic.

Achieving such prediction is a major goal of the pattern processing techniques discussed in this chapter. Patterns are identified, stored with associated contexts from other domains, and used to match against new input such that a recurrence of some pattern from the program's memory will start to spawn messages concerning its probable continuation. Another major motivation for finding and using patterns is their critical value to the composition section: transformation or generative procedures using found patterns will have a powerful, and, if the right patterns are being identified, recognizable impact on the music in progress.

6.2.1 Listening

Pattern processing in *Cypher* deals with strings of numbers. These numbers have several different interpretations: melodic intervals, harmonic progressions, rhythmic patterns, or sequences of featurespace representations can all be induced, and matched, by the same processes. All of these interpretations need some kind of preparation to get them into a numerical form which will retain enough of their essential qualities to render the resulting patterns significant. Most of this pre-processing is already provided by the normal classification functions of the listener. In the case of harmonic progressions, for example, the output of the chord and key agencies can be readily combined to produce strings of chord functions, patterns of harmonic activity which will have equal significance when transposed to any key.

In the current implementation, two interpretations of pattern strings are supported: pattern processing performs induction and matching on melodic sequences, and on harmonic progressions, where these are understood to be strings of chord functions. Featurespace words are probably at the wrong level of abstraction to yield useful patterns. There is too much information involved; good patterns are likelier to result from more restricted interpretations, such as rhythmic sequences.

The two interpretations supported are both in the harmonic domain; the real power of the technique will be greatly extended with a better treatment of rhythm. With sufficient pre-processing, useful pattern induction on durations could already be performed, I believe. The beat detection agency provides an estimate of the beat period in any given context. Incoming durations could be compared to the period, giving a duration expressed in terms of beats and fractions of beats, much as traditional music notation works.

The real power of rhythmic pattern induction, however, will be realized when more information about metric structure becomes available. At present the only rhythmic analysis done provides an estimate of the beat period, and, by extension, a prediction of the arrival time of the next beat. Finding

regularities of strong and weak beats, or meter, will require the construction of a specialized agency dedicated to the task, combining the activity of, at a minimum, the dynamic feature, and the harmonic and beat agencies. Once again we see an example of a new task made possible by the combination and extension of several known agents, which in turn are strengthened by participation in the new kinds of structure an additional agency provides.

6.2.2 Composition

On the composition side, melodic, harmonic, and rhythmic patterns recognized as active in some musical context can provide compelling material for responses from a machine performer. We saw in the previous chapter how stored sequences, or algorithmic generation processes, can provide the basis for compositional networks based on the idea of elaborating and varying material. Such networks can be strengthened significantly by adding induced patterns to the early stages of the processing. Again, later stages of transformation and adaptation to the other characteristics of an active context can mold such seed material into a convincing complement of other music. In this case, moreover, the demonstrated relation of the seed material to the rest of the context produces a response which resonates with numerous aspects of the harmonic and rhythmic flow. Once the basic processing techniques for finding and matching patterns have been reviewed, we will take up the question of how such patterns can be employed compositionally, and show some output of networks built to elaborate pattern material.

The other major contribution of pattern processing to compositional activity is the prediction of pattern continuations. As we shall see, once the pattern matcher has proceeded to a certain point with a given pattern, a match is reported as ongoing, and the continuation of it made available to the composition section. Then, compositional processes can adjust events being generated to complement the predicted continuation of the pattern. If the prediction is correct, the system will achieve cooperation with the incoming performance – anticipating, rather than just reacting to, an external musical source.

6.3 Induction

This section will describe the pattern induction process. Patterns can be induced from any number sequence; such sequences could be interpreted as representing melodies, rhythmic patterns, harmonic progressions, and so on. In *Cypher*, patterns are induced and matched for melodies and harmonic progressions.

6.3.1 Basic Algorithm

The induction algorithm began with an adaptation of David Cope's pattern matcher, described in section 6.1.1 and [Cope90]. His matcher takes two sequences of numbers, and searches through them exhaustively to find matching subsets of a length set by the user, the *patternsize*. For example, with a patternsize of three, and the input sequences { 0, 4, 3, 1 } and { 4, 3, 1, 2 }, the matcher would return the sequence { 4, 3, 1 }.

However, another of Cope's insights led to the abandonment of his matching algorithm: that is, that pattern induction is basically a matter of pattern matching, but with different kinds of data preparation, and different interpretation of the results. Following this idea, the pattern matching algorithm of Dannenberg and Bloch was made neutral enough to accommodate the demands of both induction and straight matching. There are two main advantages to this strategy: one is that a single matcher can be maintained, with the differences in processing required for induction and matching reflected in the way the process is called, and the way its return value is used. Secondly, the pattern matching algorithm used is much more efficient than the exhaustive search employed by Cope.

It is only fair to note that Cope's intention is not to develop a real-time pattern matcher. Further, his exhaustive methods will turn up patterns not found by the algorithm I now employ. However, the point of pattern processing in *Cypher* is to coordinate the induction and matching of patterns with other kinds of analysis in real time. Therefore, the constrained method developed by Dannenberg and Bloch is more appropriate to both the induction and matching of patterns in this application.

The induction process is called twice, once for each type of data which can spawn a pattern, at every phrase boundary. Phrase boundaries are demarcated in the manner explained in Chapter 3; once a boundary has been found, the harmonic progression and melodic intervals for the just completed phrase are sent to the induction process to be matched against progressions and melodies from earlier phrases.

As noted in section 6.1.3, the Bloch and Dannenberg algorithm maintains a matrix of score-to-performance matches. The basis of their technique is described thus in [Bloch&Dannenberg85]:

> An integer matrix is computed where each row corresponds to an event in the score and each column corresponds to an event in the performance. A new column is computer for each performed event. The integer computed for row *r* and column *c* answers the following question: If we were currently at the r^{th} score event and the c^{th} performance event, what would be the highest rating of any association up to the current time? The answer to this question can be computed from the answers for the previous column (the previous performance event) and from the previous row of the current column. The maximum rating up to score event *r*, performance event *c* will be at least as great as the one up to *r-1*, *c* because considering one more score event cannot reduce the number of possible matches. Similarly, the maximum rating up to *r*, *c* will be at least as great as the one up to *r*, *c-1*, where one less performance event is considered. Furthermore, if score event *r* matches performance event *c*, then the rating will be exactly one greater than the one up to *r-1*, *c-1*. [Bloch&Dannenberg85 p. 281]

Because the only information needed to compute the next column of the matrix is the previous column, the storage needed for each pattern is simply twice the largest pattern size. Further, the computation can be centered around the highest rating achieved so far, limiting the processing required to a constant as well. These characteristics of the algorithm make it eminently suitable for our real-time requirements.

The *Cypher* routine StringMatcher(s, start, newelement), where *s* is a pointer to a String (described below), *start* is the position in the matrix of the highest match, and *newelement* is the next element to be processed, is an implementation of the algorithm described above. StringMatcher is handled differently in the induction and matching processes, and given differently prepared data to deal with the harmonic progression and melodic interval cases.

To induce harmonic progressions or melodic intervals, the matcher is handed two patterns: one from the current phrase, and one from the previous phrase. The larger pattern plays the role of the performance, in the Bloch & Dannenberg sense, and the smaller pattern is used as a score. In other words, the larger pattern is matched against the smaller one. Each element from the larger pattern is successively sent to the matcher, always with the smaller pattern to be matched against. When all elements from the larger pattern have been matched, the highest rating achieved in the smaller pattern's matrix is the last element in the smaller pattern's array which was actually found. If this rating is larger than 4, that is, if at least 4 elements from the smaller pattern were also found in the larger one, the induction is successful, and an attempt is made to add a new entry to the list of known patterns.

Now, the newly induced pattern must be compared to those already known; accordingly, it is matched against all patterns already in memory. If the rating after matching the new pattern against a known pattern is 4 or more, they are considered to be the same. In that case, the known pattern's strength is incremented, and the process ends. If the new pattern does not match any known pattern, it is added to the list to be saved in the program's long-term memory.

6.3.2 Pattern Representation

Now we will briefly review the representation of a pattern in Cypher. The output of pattern induction, and the structures new input are matched against in pattern matching, both assume this form:

```
#define PATLENGTH 8
typedef struct {
        short patternsize, strength, matches;
        short element[PATLENGTH];
        short maxrating[PATLENGTH][2];
        } String;
```

Currently, patterns are held to a maximum of eight elements. There can, however, be fewer than eight: for that reason, the *patternsize* field holds the number of elements actually associated with any pattern. The *strength* field gives an indication of how many times a pattern has actually been encountered; in the section on long-term memory we shall see how a pattern's strength determines whether or not it will be preserved for future executions. *Matches* denotes the number of matches currently seen for any active pattern. Unlike the Dannenberg & Bloch algorithm, this cannot be derived from the *maxrating* matrix, and so must be recorded separately. The *elements* array holds the values against which the process will try to match. Currently, these elements could represent melodic intervals, or chord functions. Finally, *maxrating* is the matrix described in the previous section. The first dimension corresponds to one of Dannenberg & Bloch's rows, and we save two columns: one for the current event, and one for the previous event.

6.3.3 Harmonic Progression Representation

The analysis record for every event in a listening stream includes a representation of the event's harmonic function relative to the active key. As each successive event is read from the stream and added to the phrase, the root of the local harmonic area, and the current key, are obtained from their respective agencies. With these two pieces of information, a simple calculation provides the function. Both chords and keys are stored as an integer between 0 and 23. Major and minor modes of the same root are stored adjacently; chord and key number zero are both c major. Therefore, for both chord and key, c major is 0, c minor is 1, c# major is 2, c# minor 3, and so on. With this system, we can use the following define statement to calculate chord function within the key:

```
#define Function(key, chord) (((chord+24)-((key%2)?key-1:key))%24)
```

The effect of the defined Function statement is to move the position of zero across the collection of chord theories. If both key and chord are already zero, for example, the position of zero in the chord theories will remain on c major, and indicate that chord as the tonic of the key. If the key is 4 (D major) and the chord 8 (E major), Function will return a value 4, indicating a major chord based on the scale degree a major second above the tonic. If the key is minor, however, Function subtracts the value of the major key on the same root: this keeps minor functions on odd values, and major functions represented as even values, the convention for all harmonic reports in the program. Relating local harmonies to the key effectively removes key from consideration in storing and matching harmonic progressions. A functional progression of tonic to dominant to tonic will look identical, in this representation, regardless of the key in which it was originally presented.

The analysis record for every event in a stream has a representation of the chord function included in it. Chord progressions are sent through the induction process at each phrase boundary; therefore, when a phrase boundary is found, a list of the chord functions is made by reading from the beginning of the just completed phrase to the end, appending to the list function values for every event along the way. If a function for some event is found to be identical to the one before, it is discarded.

6.3.4 Melodic Representation

The other source of sequences currently being scanned by pattern induction is melodic activity. Melodies are strings of single-note events; the phrase boundary demarcation again is used to identify melodic groups to be submitted to the pattern inducer. Preprocessing of melodic information into a pattern requires two steps: first, we are currently considering only monophonic melodies, so as the pre-processor steps through the events in a phrase, events with a vertical density greater than 1 are discarded. Second, as in the case of harmonic progressions, it is desirable to give the pattern some degree of independence from the specific context in which it is embedded.

In the case of melodies, this is done by casting specific pitch sequences into sequences of intervals. Each new pitch appended to the melody list is compared with the previous one: the difference between the two note values is the value added to the list. So, a rise of a major third from C3 to E3 (MIDI notes 60 to 64) is represented as +4, an ascending major third. Descending intervals will produce negative intervals. Intervallic representation allows melodic contours beginning on different scale degrees, or in different registers, to be successfully matched. Note that this representation means that there will be one fewer interval generated than the total number of events in the phrase.

6.3.5 Long-Term Memory

As patterns are induced, they are added to a list of known patterns. Two lists are maintained: one for melodic patterns, and the other for harmonic progressions. Every time *Cypher* ceases operation, it writes out a file storing a record of the patterns it knows. This file represents the long-term memory of the program. Then, whenever *Cypher* starts up again, it loads in the patterns from long-term memory and activates them for matching. In this way, the program incrementally learns and uses patterns found in the music played to it.

Each known pattern has an associated strength: the strength is an indication of the frequency with which the pattern has been encountered in recent invocations of the program. Whenever the inducer notices a pattern, it attempts to add the pattern to the known list. Before such an addition will succeed, the pattern is matched against those already on the list. If the pattern is similar to one already known, that is, if it matches most of the known pattern, the new copy will not be appended. However, in that case, the strength of the already stored pattern will be increased.

Pattern strengths are used for two purposes: first, strong patterns will be used more often by the composition section as seed material. Second, pattern strength decays slightly with every program execution. If a pattern is not seen again through a sufficient number of executions, it will be deleted from the

list. In other words, if a pattern is learned, but then not seen again for a long time, it will be forgotten: dropped from the long-term memory file.

6.3.6 Evaluation

The pattern induction process I have just described works, after a fashion: it has successfully found chord progressions and simple melodies from listening to music. These successes, however, are limited to what I would call "toy" examples. Induced chord progressions, for instance, are found by repeating the progression literally several times in succession until the program notices it. Similarly for melodies, simple sets of intervals not more than eight notes long can be found by pattern induction if they are repeatedly played to the machine.

Much more desirable, of course, would be pattern induction able to function in the real world. Finding chord progressions and melodic patterns from improvisation or performance of compositions not geared to the algorithm, is where this work is intended to lead. The reasons such "real world" functionality remains elusive, in fact, have much less to do with finding an efficient matching algorithm, than they do with good preparation of data to present the matcher as input. There are at least three major problems here: 1) building patterns on the right level; 2) successfully grouping information into patterns; and 3) storing good candidates for repeated matching attempts.

Different pattern types will present meaningful sequences of data on different levels. Melodic information, for example, will most often be found on level 1, in the succession of events. However, even though the first place to look is on the event level, some evaluation of the data should be done before building a candidate pattern to do things like remove ornaments. A "real world" pattern inducer should be able to find melodies repeated with some ornamentation. Having a speed feature classification at the grace note level would be a first approximation: events with the highest speed rating could be discarded when building melodic patterns for induction.

Harmonic progressions, however, are most likely not found on the event level. Rather than trying to match chord sequences presented in a phrase, the

progression preparation should try to identify an important, or dominant, chord or two within each phrase. Sequences of important chords could then be built into patterns and presented as candidates for induction.

The second problem, finding good groups, is related to the workings of the phrase boundary agency. As it is now, induction takes place at the end of each phrase, comparing the just completed phrase with the one before it. This works reasonably well, as long as the phrase boundary finder is reliably finding boundaries in the same place, when presented with similar music. Particularly before the phrase threshold has stabilized, this is not always the case. Perfectly plausible patterns, then, may not be found, because a spurious phrase boundary would interrupt the construction of exactly the pattern we might wish to notice. I believe that the phrase boundaries *Cypher* finds, give critical information for finding meaningful patterns. However, further grouping rules will have to be devised, to ensure that melodies spanning phrases, or the harmonic progressions built from prominent chords, will be built into patterns and sent through induction.

The third problem involves the question of how to identify sequences which are worth saving for future matching attempts. In "real world" music, interesting patterns will rarely be literally repeated end-to-end. Normally, some melody, for example, will appear and re-appear, but with each occurrence separated from one another in time. In a pattern inducer looking for matches between adjacent phrases, such sequences will not be matched. What is needed is some way to select melodic or harmonic strings as interesting in their own right, such that they will be kept around and matched against future sequences even when an immediate match is not found. This is the main lesson I learned in implementing the induction process reported here: repetition alone is an insufficient heuristic for finding good sequences. Melodies and harmonies must also be recognized as interesting because of some intrinsic qualities, and tried out repeatedly for matches against subsequent material. It may be that the analysis *Cypher* performs is enough to mark interesting strings; I do not wish to claim I have found such a method here, but rather to identify a necessary enhancement of the technique I am reporting.

6.3.7 Getting the Blues

A good style for testing ideas of pattern induction and matching is the blues, particularly in terms of harmonic progressions. It was in the process of playing blues examples, in fact, that the possibilities and, above all, weaknesses of the pattern processing implemented to date, were brought to light. The blues are characterized by relatively straightforward chord progressions, seemingly a good place to try induction. And, indeed, if the progressions are played in their most basic form, induction succeeds. If real blues examples are performed, however, pattern induction rarely finds useful sequences.

The reasons for this failure were enumerated in the preceding section: let us briefly examine how addressing those problems could lead to a successful blues player. The first problem mentioned above, finding the right level to look for patterns, may be the most nettlesome here. The right level for chord progressions would seem to be somewhere between the current chord and key analyses. The chord agency returns a new value for every event; this is too fast. The key agency usually remains stable to long sections of a piece: only a few key changes will be recorded to any given tonal composition. That level of description, then, is too slow.

Identifying important chords in a phrase would seem to yield the right level for inducing chord progressions. In the simplest forms of the blues, the underlying harmony will change not more than once per phrase. If the dominant, or characteristic, chord for each phrase can be isolated, these strung together would be the sequences appropriate for pattern induction.

The second problem, finding good groups, is related to the level problem just discussed. The phrase identification process implemented in *Cypher* will not work for a level of harmonic description between chords and keys. An even higher level of analysis, gathering together the characteristic chords of groups of phrases, would be the appropriate tool for segregating meaningful chord chains.

The third problem might not be too serious for blues examples. If chord progressions at the level of description described above were isolated, there is actually a good chance that they will be repeated quite regularly. Rather than an additional algorithm to mark some progressions as interesting, then, induction might succeed simply through attempting to match against two or three of the most recent sequences.

6.4 Matching

The pattern matching algorithm described in section 6.3.1, in connection with the induction process, is the same one used for the related problem of matching induced patterns against incoming data. StringMatcher(), the routine responsible for matching patterns and returning a rating of the highest matched element between them, is the same one used here. In this section, I will review the differences in the way data is prepared, and results interpreted, to handle the matching problem.

Because of the properties of StringMatcher(), every incoming event can be tested against the known patterns: there is no need to wait for a phrase boundary to find groups, as in the case of pattern induction. When the rating of a known pattern becomes non-zero, after exposure to incoming events, some part of it has been matched. Therefore, the general strategy is this: when a new event arrives, it is matched melodically and harmonically against known patterns. If more than half of a known pattern is found, a message is sent to the composition section along with the remainder of the pattern. If all of a pattern is recognized, another message is sent to the player, the rating matrix is reset to zero, and the process begins again.

6.4.1 Melodic Matching

Though the matrix rating algorithm of Dannenberg and Bloch has proved to be a useful starting point, significant adaptations have been made to it for the tasks I describe in this chapter. The algorithm has been changed, because our demands are significantly different. For example, in the score following case, an assumption is made that the performance is unlikely to go backwards. That is, once the matcher is relatively sure of having matched some part of a

score, it will try to continue on from that point. In matching melodic patterns, however, we can never be sure when a pattern might start, and have less reason to believe that matching any part of it means we will not see that part again before the rest of the pattern. Therefore, the ratings of the original algorithm have to be modified if a partial match has not been continued for some time.

One of the efficiencies of the Bloch and Dannenberg algorithm is that a window can be centered on the highest rated event found with the matrix, and continued from there. The same convenience is adopted here, as is evidenced by the *start* argument to the StringMatcher() routine. The deviations from the Bloch and Dannenberg work come when no new match is found. In that case, one is subtracted from the rating of all of the elements in the matrix. Our window begins at the point of the latest match, represented by the *start* argument. However, it continues from there to the end of the pattern. Therefore, when we subtract one from all ratings on an unsuccessful match, we effectively start looking from one element further back in the target sequence with the next match. If the next element is found on a subsequent attempt, nothing is lost – because we continue to search to the end, it will match in the same place as if no decay of the ratings had taken place. If the next element is not found for several events, however, the decay in ratings means that the matcher will again be searching from the beginning of the pattern. In other words, if the continuation of some pattern is not detected, the decay in element ratings will gradually return the matcher to the start condition, where it is again looking for the head of the sequence.

6.4.2 Harmonic Matching

The harmonic matching case is a close analog of melodic matching. The differences arise in the preparation of sequences, and the response made to matches. As we have seen, the melodic matcher is sent intervals found between single-note events. In harmonic matching, the chord function of new events is sent on to the matcher to be compared with known harmonic progressions. At present, there is no harmonic equivalent to the composition method FinishMelody(), the routine called when a partial melodic match has been found. Responses to partial harmonic matches can be easily envisioned:

for example, the remainder of a progression could be forwarded to the critic, which would adjust incipient output to accord with predicted harmonic functions (through the *Harmonizer* transformation).

6.5 Compositional Elaboration of Patterns

In Chapter 4, I described compositional networks built around the idea of stages of processing. Generally, algorithms which generate material are invoked in the early stages of such a network, and other transformations of that material are invoked later, adapting the output to the demands of the musical context. Patterns learned by the processing described in this chapter are well suited to use in such networks, particularly as material inserted at the beginning of the chain. Melodic patterns, for example, are excellent candidates for transformation, preserving recognizable elements of a linear contour, which is nevertheless adapted to the harmonic, rhythmic, and other featural demands of the music surrounding it.

Another important compositional strategy is to use the remainder of a partially matched pattern. We have seen that the matcher returns a rating, representing the number of matches of the new pattern against a known one. Because of the way this matcher uses the rating matrix, the first location in the matrix of the maximum rating reached by the match, equals the position of the last matched element in the known pattern. With this information, the composition section can schedule the rest of the pattern for performance, or for further elaboration from a compositional network.

The routine FinishMelody(String *s), for example, is called when the rating and number of matches returned for some pattern is greater than half its length. FinishMelody() will then examine the *maxrating* array to find the last matched element. The rest of the elements after that last match, will be scheduled to play on successive beats after the invocation of FinishMelody(). This is a crude but effective way to test the matcher: a frame representation storing the number of beats associated with each remaining element of the pattern would preserve the rhythmic integrity of remembered material. Still, playing out the remainder on the beat is already much to be preferred to playing it back at an unchanging speed whenever the beginning is found.

With other transformations to improve the relation to the current context, even an un-framed collection of intervals can be useful compositionally.

Harmonic sequences can similarly be elaborated in several ways. Their natural position in compositional transformation chains, in contrast to the melodic material, will be towards the end. In other words, after some seed material has been fed through a number of elaborations, one of the last steps in the process is often to make sure the resulting material is consonant with some overall harmonic design. Usually, the *harmonizer* transformation does this, consulting the chord and key listener reports and adjusting pitch material in the output block accordingly. With chord progression patterns, the harmonic activity can become more directed: rather than staying consonant with some external context, the composition section will know which chord is coming next, enabling it to not only accord with the current harmony, but anticipate the one coming up. A new version of *harmonizer* could be written to provide pitch transitions leading from one chord function to another – in effect, a machine implementation of voice leading.

Chapter 7

Score Orientation

In performing pieces of composed music, the computer performer must be able to change its overall behavior at the appropriate moments in the score. One way to think of this is as very high level tracking, looking for major articulations of the piece which call for new kinds of response. However, there are two main distinctions between the high level analysis techniques we have examined so far, and the necessity of advancing through a script of responses during the performance of a musical work. The first difference is that points of state change are indicated by the composer; not the phrase boundaries determined by the program, but points in the score chosen arbitrarily by the composer must be found. Secondly, the kind of analysis needed to locate such cues is primarily one of pattern matching; in most cases, the music of the human performers will be sufficiently well determined to identify unambiguously when a cue has arrived. In this chapter, we will discuss techniques for coordinating state changes with a performed score, processing referred to here as *score orientation.*

I begin by differentiating score orientation from the related technique of *score following*. Then, I will describe the specific facilities available in *Cypher* for navigating through a score of composed music. Finally, I discuss rehearsal techniques for integrating the computer part with a human ensemble, and describe the graphic interface which enables this process.

7.1 Contrast with Score Following

Score followers, a class of applications which have been significantly advanced by Barry Vercoe [Vercoe84], Roger Dannenberg [Dannenberg89], and Miller Puckette (among others), match a live performance against an internal representation of the performed part as the program expects to hear it. The goal is to determine the tempo of a human performance. A pattern matching process directs the search of a space of tempo theories. The derived tempo is used to schedule events for an accompanying part, played by the computer, whose rhythmic presentation follows the live performer – adapting its tempo

to that of its human partner. A comparison of the time offsets between real events, and the time offsets between stored events (to which the matching process has decided that the real events correspond), yields an estimate of the tempo of the real performance, and allows the machine performer to adjust its speed of scheduling accordingly.

An important part of score following applications is storing a representation of the expected performance such that it can be efficiently be matched in real time. Another critical component is the pattern matcher itself, which must accept performances containing unforeseeable deviations from the stored version, and be able to find the correct point in the score after a period of instability in the match.

Score orientation, therefore, differs from score following in those two critical areas. In score orientation, the program is following the human performance to find those points in the composition where it is to advance from one state to the next. The technique does not follow a complete representation of the human player's part, but is only scanning its input for cues at those moments when it expects to perform a state change. Further, score orientation can be accomplished with much less sophisticated pattern matching. Cues can usually be found which will be unambiguous enough inside a given time frame that the orienter can simply search for that single event. Better matching will improve the performance of score orientation, but is not critical to the success of the method.

7.2 Windowing

Cypher's score orientation process uses a technique called *windowing*. A window is a designated span of time. During the duration identified by a window, the program is searching for a particular configuration of events; when that configuration is found, the program state is updated, and, if necessary, the opening of another window is scheduled. If the desired configuration has not been found by the end of a window, the state change and scheduling of the next window are done anyway. This ensures that the program will never remain in a given state indefinitely waiting for any single cue.

Cues can be missed for a variety of reasons; in practice, one missed target will not throw off the orientation. Usually, the next window successfully finds a match near its leading edge. If two or three windows in a row pass without a match, however, the orienter is irretrievably lost, and must be righted through external intervention. But even if nothing were ever played to the score orienter, the program would eventually make it to the end of the piece – it would just no longer be synchronized with an external performer. The graphic interface described later in this chapter provides mechanisms for beginning execution at any given state, and for recovering from missed cues, if this should be necessary in performance.

There are, then, six parameters associated with a score orientation window: 1) the time offset between the identification of the preceding orientation cue and the expected arrival of the next one; 2) the leading edge of the window, or time in advance of the expected arrival of the target event, at which to begin looking for it; 3) the trailing edge of the window, the duration after the expected arrival of the target event during which to continue looking for it; 4) the type of event being targeted (pitch, attack, pattern, etc.); 5) the specific event of that type to be located (pitch number, attack point, pattern description, etc.); and 6) the routine to be invoked when a target has been found. All of these parameters are coded by the user into a cue sheet, which the score orientation process will use to schedule and execute polling for the designated events during performance.

7.2.1 Cue Types

There are six different cue types for events targeted by the score orientation routine. These are:

• **Pitch**. Any MIDI pitch number can be specified as the target event.

• **Attack**. At the opening of the window, a pointer is set to the most recent incoming MIDI event; the next event past this pointer, that is, the next event played, will trigger the state change.

• **Root**. A chord root, expressed in absolute terms (C, F, Ab, etc.) as opposed to relative terms (I, IV, V, etc.) can used as a target event.

• **Function**. Chord roots described as functions, that is, in relative terms (I, IV, V, etc.) can be used as targets with this event type.

• **Time**. This option essentially bypasses the windowing mechanism. The state change will be executed when the duration indicated has elapsed, regardless of other conditions.

• **No Attack**. This type uses the Boredom parameter; when no MIDI input has come for a duration exceeding the Boredom limit, *no-attack* type state changes will be executed.

This collection of cue types reflects the demands of the compositions learned by this implementation of *Cypher*. These six have the virtue of simplicity, and are quite robust. However, they are by no means an exhaustive list of the types of cues one might want, or even of the cue types the program could easily accommodate. In essence, any of the messages coming from a listener could be used as a cue; a certain register identification, for example, or a phrase boundary, could be recognized by the score orientation process and used to trigger a state change. It is a trivial extension to the program to include other cue types, and to search for specific targets of those types. The list above should be regarded as an example, motivated by the circumstance of a few compositions; the complete list would be simply an inventory of all possible listener messages.

7.2.2 An Example

The following example is taken from *Flood Gate*, a composition for violin, piano, and *Cypher*, described in section 8.1.1. In this example, we will examine a score orientation state change, associated with a pitch cue. In figure 43, at the beginning of measure 6, the *A* circled in the piano part is the target of the window associated with state 1. The opening of the window arrives some seconds before measure 6, approximately one measure earlier. The circled *A* is chosen as a target for two reasons: first, because it is an important

event in the context of measure 6, one likely to be treated carefully by the pianist in any event; and, second, because there are no other appearances of that pitch in the immediate vicinity – particularly, none in measure 5, when the window opens.

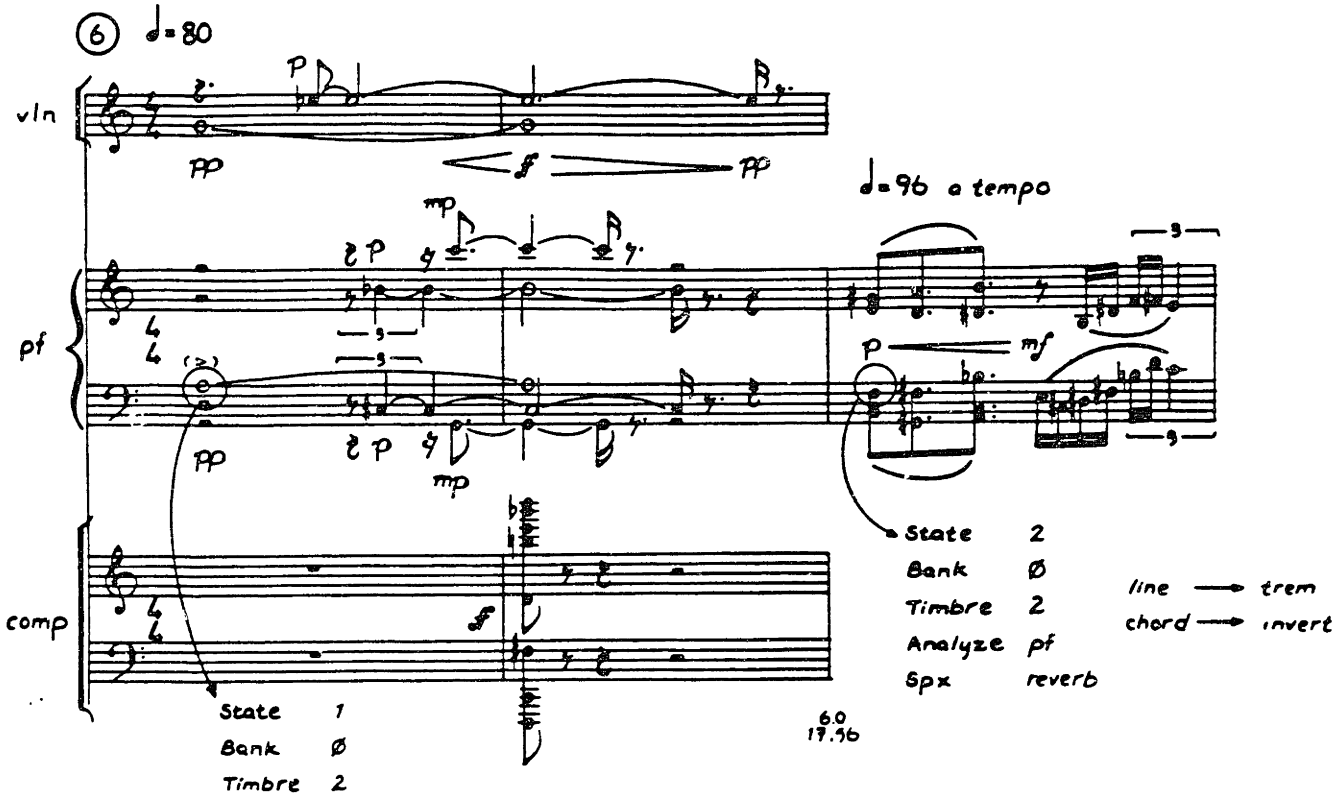


Fig. 43

The expected arrival time of the *A* is the focal point of the score orientation window for state 1. The estimate of when this cue will arrive is made at the time of the previous state change, that associated with state 0. When state 0 is activated, the current absolute clock time is noted. The ideal temporal offset of the next cue, that associated with state 1, from the time of activation of state 0, is added to the current time. The result is used as the estimated time of arrival of the following event. When the cue for state 1 arrives, again an ideal offset preceding state 2 is added to the current clock time.

Ideal cue arrival times are initially calculated from an examination of the score. Using the notated metronome marks as a guide, the temporal offset between one point in the score and another can be determined from the number of beats at a given tempo separating the two points. The rehearsal process will serve to bring these ideal estimates in line with the actual performance practice of a particular human ensemble. The offsets between cue points, and the window sizes surrounding each cue, will typically be tuned during rehearsal to find a comfortable fit between *Cypher*'s score orienter and the other players.

Returning to our example, the window associated with state change 1 remains open until approximately the beginning of measure 7. When the expected *A* is found, the chord shown in the computer part at the beginning of measure 7 is scheduled, as is the opening of the next window, associated with state 2, targeted for the *F* at the start of measure 8. The chord, which is notated in the score at the downbeat of measure 7, will be scheduled as an ideal offset from the arrival of the cue pitch A; in other words, if measure 6 is played exactly in tempo, the chord will arrive exactly on the downbeat of measure 7. However, the gesture is composed to allow for the arrival of the computer's chord somewhere other than exactly on the beat: the musical idea is a swell in the violin and piano leading to a dynamic climax accentuated by the computer part. In some cases, even with score orientation, the computer takes the lead: this is a natural example, in that the arrival of the chord marks the high point of the crescendo. The human players quite naturally adjust their performance to accommodate their machine partner's timing.

If the end of the window associated with state 1 is reached without finding the *A*, that is, if we reach measure 7 without seeing the target, the chord and state 2 window opening are scheduled anyway. In that case, the chord opening measure 7 will be quite late; usually a very telling signal to the human players that Cypher is confused. However, the *F* of state 2 will then probably be found close to the start of the window for state 2 (since the window scheduling was initiated a little late), and the computer and human performers are then once again synchronized.

7.3 Rehearsal Mechanisms

One of the most critical components of an interactive system for composed music is a rehearsal mechanism. Artificial performers should be able to integrate with normal rehearsal techniques as seamlessly as possible. This demands two things of the program: that it be able to start anywhere in the score, and be able to get there quickly. Musicians tend to start at structural breaks in the music, and may start at a certain bar several times in rapid succession. Or they may start from an arbitrary point and play through the rest of the piece. If it takes the computer markedly longer to be prepared to start playing from any point in the score, human players' patience with their neophyte performance partner may flag at the most critical juncture, during rehearsal.

In *Cypher*, the performance file for a composed piece is made up of complete state records for every change of state in the composition. These states may be accessed through two modes, marked on the interface *Single* and *Free*. Both these options reference a state slider to find out what the initial state number for the rehearsed material should be. *Single* will return the program to that state, and stay there. This mode allows rehearsal of small sections of the score for as long as is desired, without advancing to subsequent score states. In contrast, *Free* will revert to the state number given by the slider, and commence performance of the entire piece from that point. In this mode, the succession of state changes will be executed as in a complete performance of the work, allowing rehearsals to start at any given point and continue from there.

In both modes, selecting a state means that the window for finding that state's target event is opened. In other words, calling for a free run of *Flood Gate* from state 1 will open a score orientation window waiting for the *A* in the piano part at measure 6 associated with that state (as described in the section above). Actual execution of the state change will take place when the pitch is played, ensuring that the subsequent computer and human performances will be synchronized.

7.3.1 Saving and Restoring States

Before rehearsal begins, naturally, the states associated with each point in the score must be recorded. *Cypher* offers facilities to save and restore states from a single file, such that one file can be used for each separate composition. A state record is stored in about 2K bytes. Collections of state records are stored in a single file on disk, and can be accessed at will from the graphic interface or by processes at work in the program itself.

In a studio setting, then, the composer can manipulate graphic controls to arrive at a mapping of input features to output transformations deemed worthy of preservation. The state record is saved to a file under a state number. As many states as desired can be saved in a single file, and later associated with target points for performance by score orientation. The studio work of finding useful configurations can be recalled onstage either through the score orientation method, as mentioned, or through the graphic manipulation of the state files by a human operator.

7.3.2 Program State

A *state* is the set of variables whose values uniquely determine the function of the program. To change overall behavior, the program must advance from one state to another during performance. The state of *Cypher* is described with the following variables:

- **Sounds.** The bank, timbre, and processing settings (section 7.5.1).

• **Connections**. The relations between level1 and level2 listener and player agents.

• **Attention**. The MIDI stream being analyzed by the program, either an external source or *Cypher* itself. If the stream is an external source, the channel number being tracked is stored.

• **Boredom**. The duration of time the program will wait before beginning to produce output introspectively.

7.4 Random Access Orientation

Windowing is a fairly reliable technique for score orientation, which has been used successfully in scores of live performances. Most often, the orientation process correctly identifies all of the cue points in concert. Sometimes, cue points are missed; if two cues in succession are not found, operator intervention is almost always required to get the orientation process back on track.

The success the orienter has is due in large measure to the continual adjustment of the window parameters during the rehearsal phase. Initial window offsets are calculated by assuming a performance which follows tempo indications exactly. Rehearsal reveals rather quickly the real tempi, which can be used to adjust the cue timings. Further, rehearsal will show the optimal window sizes, which depend on such things as the presence of non-cue targets in the vicinity of the real cue, the length of time between a cue and the next, etc.

A significant advance in score orientation would be achieved if the process were able to modify window parameters itself during rehearsal. However, the current implementation is not sophisticated enough to build such a facility. The greatest problem is that when a window ends without locating the target event, the orienter has no way to tell whether the cue was not found because it was played before the window opened, or because it has yet to be played, or because an error in the performance means that it will never be played.

The cure for this is to provide the orienter with better targets - not single pitches or attacks, but small local contexts. If most of the recognizable information from a whole measure is provided, for instance, missing one note within the measure would have no effect on the orienter.

7.4.1 Pattern Matching

The relative instability of score orientation is a product of the small size of the target events used. Basing a state change on a single event will be uncertain no matter how accurate the transducer transmitting the events. Once the targets being tracked gain the stability of more information, orientation should improve markedly. The only disadvantage, in current hardware environments, is that the amount of processing necessary for matching the targets will also rise significantly.

In effect what is needed is a score follower for small fragments of music, which is not trying to extract tempo, but merely to establish where the human performers are in the score. Currently, the windowing technique looks for certain events around certain times. With very fast pattern matching, a small pattern, for example one measure of the score, could be associated with each orientation point. When performance of some composition is selected, all of the score orientation fragments are made available for matching. If the matcher becomes reasonably certain it has found one of them, the score operations associated with that target pattern can be performed.

In an ideal implementation, windowing would become superfluous. The program then looks for all the change points all of the time. A bias towards finding points in succession would be appropriate, but looking for all points simultaneously would solve the rehearsal problem, for example, since the players could start anywhere and be found by the computer, and would allow the performance of indeterminate scores or improvisations, whose order is not known in advance, to be handled by the same mechanism.

7.5 Interface

Cypher is equipped with a graphic interface to allow the connection of features to transformation modules, the selection of timbres, and the preservation and restoration of states. A reproduction of the control panel is shown below :

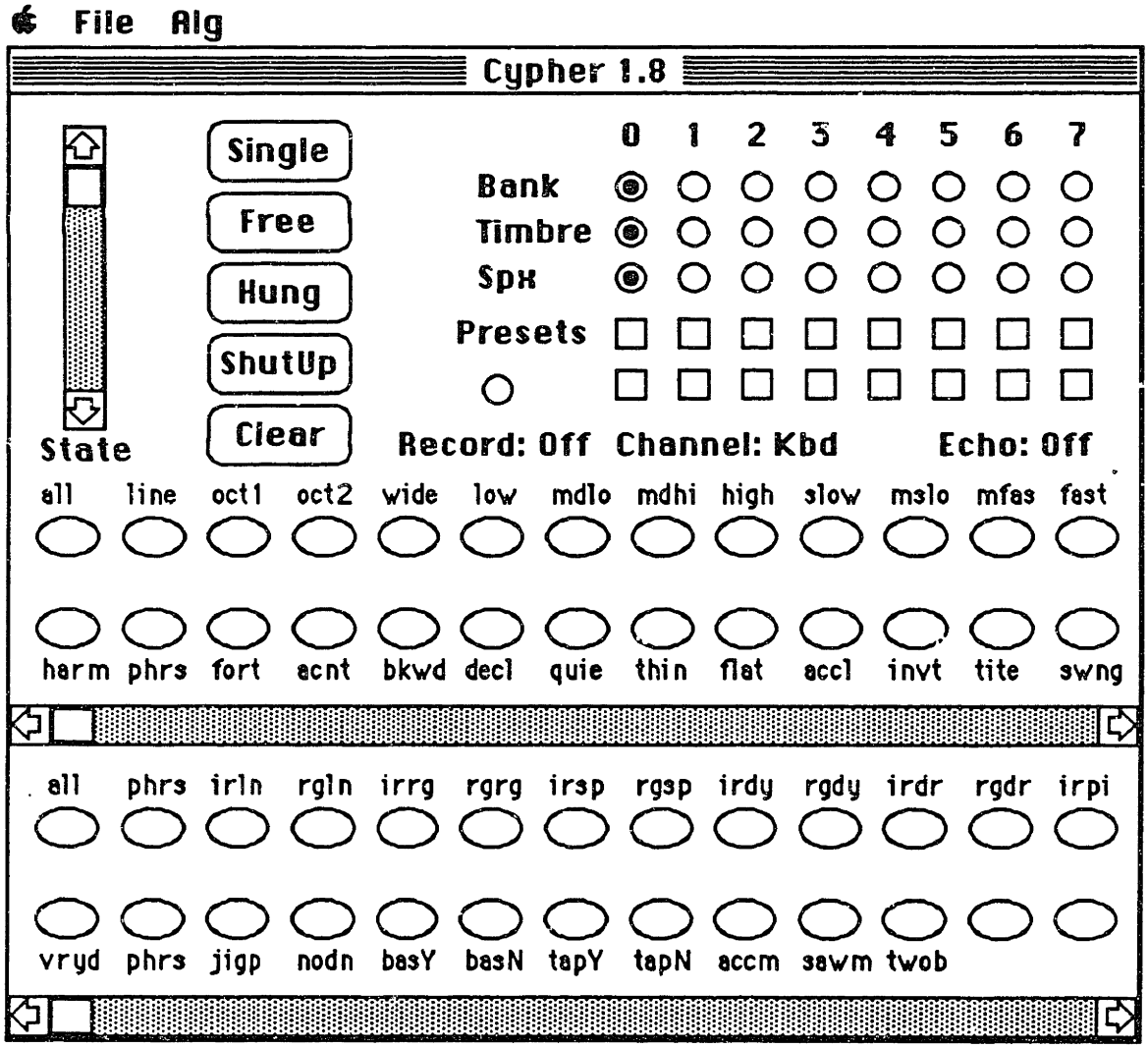


Fig. 44

7.5.1 Timbre Selection

The set of buttons in the upper right hand corner of the display is used to select voice banks, and subsets of those banks, from the performance devices connected to the computer. These buttons can select only one of the choices at a time: for instance, the *Spx* buttons allow the user to send one of eight possible program changes to an SPX-90 digital signal processor. The *Bank* and *Timbre* buttons together determine the sounds used to play the output of the composition section. The meaning of these buttons is established in a text file, called *cypher.voices*, which is read by the program on startup. This text file allows a user to change the effect of the *Bank* and *Timbre* buttons simply by editing *cypher.voices*, to match any local configuration of synthesis gear; no further modification to the program is required.

```
10                                      ; number of output channels

0 1 2 3 4 5 6 7 9 10                    ; output channels to use

16 11  8 29  3  9  1  3  1 21  ; bank0 programs
18 11 17  2 28 13 30  2 12  2  ; bank1 programs
24  8 15 26 16 30  6 12  4 20  ; bank2 programs
 8  1 12 19  1 24  9 14  6 16  ; bank3 programs
 1  5 26 32 30 19 14 16  7 18  ; bank4 programs
 7  9 12  1 31  9 30 11 14  3  ; bank5 programs
 7  3  8  1 16  7  8 18  5 19  ; bank6 programs
23  4 17 14 31  9  9 24 17 15  ; bank7 programs

3 2 3 4 4 3 4 5                         ; # of channels per timbre

2  7  9                                 ; output channels of timbre 0
7  9                                    ; output channels of timbre 1
1  5 10                                 ; output channels of timbre 2
0  2  3 10                              ; output channels of timbre 3
0  1  2  9                              ; output channels of timbre 4
3  9 10                                 ; output channels of timbre 5
4  5  9 10                              ; output channels of timbre 6
0  4  5  6  9                           ; output channels of timbre 7

0 1                                     ; input channels
```

A listing of a typical *cypher.voices* file is shown above. The first configuration specification recorded, as we see, is the number of MIDI channels available for output. In this configuration, there are ten MIDI output channels. Next, the file indicates which channel numbers are associated with the ten outputs.

Then, a series of program changes are listed. Here, the meaning of the *Bank* buttons is established. Each *Bank* button calls up a set of program changes, one for each output channel. Which program changes are sent is specified as shown in the listing; there are eight sets of program changes, one for each button on the interface. Using this configuration file, when a user clicks on *Bank* button 0, MIDI channels 0,1,2,3,4,5,6,7,9, and 10 will receive the program change messages 16,11,8,29,3,9,1,3,1, and 21 respectively.

The *Timbre* buttons choose subsets of the nine possible channels, which will all receive copies of the MIDI note on and note off messages coming from the composition section. The entries in *cypher.voices* shown above after the specification of the *Bank* buttons deal with the meaning of the *Timbre* buttons. First, the number of channels across which output will be sent is listed, one entry for each *Timbre* button. Thus, following this specification, *Timbre* button 0 will instruct the program to send out on three channels, *Timbre* button 1 on only two channels, and so on. Next, the configuration file lists precisely which output channels those are. So, for *Timbre* button 0, we know that it will limit output to three channels. In the line followed by the comment `"output channels of timbre 0"` we see that those three will be channels 2, 7, and 9.

Finally, two entries indicate which input channels will be monitored by the program when either the *Kbd* or *Ivl* selections are chosen from the interface. *Kbd* and *Ivl* refer to the two most commonly used source of MIDI input signals: either some kind of MIDI keyboard, for example a normal keyboard controller, or MIDI percussion controller; or an IVL pitchrider, a commercially available device that converts audio signals to streams of MIDI data. On the interface, there is an entry marked **Channel:** followed by either *Kbd, Ivl,* or *Nobody*. Clicking on the current selection will cycle through the three possibilities. When the **Channel:** selection reads *Kbd, Cypher* will listen to events arriving from the channel listed as the next-to-last entry in *cypher.voices*. When **Channel:** selects *Ivl, Cypher* listens to MIDI coming from the channel listed in the last *cypher.voices* entry, and when **Channel:** selects *Nobody, Cypher* will not respond to MIDI events arriving from any channel.

7.5.2 State Selection

The buttons and state slider in the upper left on the control panel make up the state save and restore mechanism. The slider is used to select a state record number; these range from zero to sixty, though the upper limit is completely arbitrary and easy to change. To record a state to memory, the user first positions the slider so that the state number showing beneath it corresponds to the desired state record in the file. Then the item *Save* is selected from the *File* menu. This will record the current state shown on the interface to the currently open file, at the given state record number. If no state file is open, a dialog box will first appear asking which file is meant to receive the new record. The complementary operation is performed in response to the *Restore* item in the *File* menu. Again the record number is accessed from the slider, and then the requested state is retrieved from the open file. If no state file is open, a dialog box asks for the specification of one. If no such record is present in the open file, an error message is generated.

7.5.3 Connections

There are four rows of ovals present on the interface. These represent listener and player processes on levels 1 and 2. The uppermost row of ovals represents the first level of the listener, and the next lowest row of ovals, the level 1 player. The listener ovals show the features which can be extracted from single events (there are more than one screen can show – scrolling in the horizontal bar would bring the rest into view). The player ovals show the possible transformations (again there are more out of view). To establish a connection between a feature and a transformation module, the user need only draw a line connecting two ovals with the mouse.

When a feature and transformation have been connected, any input event which is classified as having that feature will be transformed by the connected player module, and the result played out. All transformations tied to all features found in the input event will be applied to that input before the result is output from the composition section. If several features are attached to transformations, all of which are to be invoked, the set of transformations is applied in series, with the output of one being fed through to the input of

the next, as described in section 4.2. Any feature/filter connection may be broken again by clicking once in an oval at either end of the line. In fact, clicking on an oval will disconnect all the lines arriving at that oval, so that if several transformations are all connected to a single feature, clicking on that feature will disconnect all the transformations. However, clicking on the transformation end of the line of that connection will only disable the link between the feature and that single transformation. Any other links to the same transformation, however, will likewise be severed.

A feature may be connected to any number of transformation modules; similarly, a transformation module may be connected to any number of features. The first oval in the feature collection is marked *All*, which means that the transformation modules to which it is connected will be applied to all of the input events, regardless of their features.

As we reviewed earlier, the application order of transformations will affect the result. Because they are invoked in series, the same collection of transformations will produce different outputs, depending on the order of their execution. The priorities of the transformations are reflected and established by their appearance on the interface. When several modules are to be applied to an input event, they are invoked from right to left as they appear on the screen. To reorder the priorities, a module can be "picked up" with the mouse and moved right or left, giving it a lower or higher priority relative to the other transformations. This new ordering of transformation priorities will be stored in the program's "long-term memory", so that the next invocation of the program will still reflect the most recent ordering.

The third and fourth rows of ovals represent the listener and player on the second level. The listener ovals (again, those higher on the panel) refer to the regularity and irregularity of the first level features: *irrg*, for instance, stands for *irregular register*, meaning that the registers of incoming events are changing within phrases. The complementary analysis is indicated by *rgrg*, which stands for *regular register*, and means that the events in the current phrase are staying in a constant register more often than not. From left to right, the features which are tracked for regularity are density, register, speed, dynamic, duration, and pitch. The lowest row of ovals represents level2

player processes, which do things like send *mutate* messages to the level1 transformations, or create and delete connections on level1. The connection (or disconnection) of regularity detectors to player processes can again be accomplished graphically.

The *Presets* check boxes are simply collections of connections; they are toggles, which alternate between making and breaking certain sets of connections. As many as desired can be turned on at any one time. Currently, the connections called up by the *Presets* boxes are specified in code, and are compiled into the program. Ideally, a configuration of listener/player connections should be saveable from the interface to any of the presets.

7.5.4 Menus

The *File* menu provides a number of commands which have to do with the state save and restore mechanism, reading MIDI files from sequencers or other software, and the execution of *Cypher* itself. The first three menu items have to do with opening and closing state files. These items follow the layout of the usual Macintosh application *File* menu: *New*, *Open*, and *Close*. The *New* command opens a new state file, which will then be ready to receive and retrieve state records. *Open* is similar, but will present a dialog box with the names of all currently available state files, from which the user may choose. Finally, *Close* will close any currently open state file, preparing the way for a new one to be used.

The next *File* menu item calls up a dialog box which allows the user to play standard MIDI file sequences into *Cypher*. The data read from the MIDI file will be treated as if it had been played from an external MIDI instrument in real time; that is, each event will be evaluated by the listener, provoking transformations and response by the associated player processes. See the section in Chapter 4 on sequencing techniques for a more thorough description of this facility.

The next set of entries on the *File* menu refer to the state save and restore mechanism, described above. *Save* will save the current program state to a file, and *Restore* will restore a state from an open file. The *Dump* command

instructs the program to save each state, when passing through states with the score orientation process. When *Dump* is turned on, a complete state file for a given composition will be saved automatically. The final *File* menu entry is *Quit*, which exits the program.

The other available menu is the *Alg* (for Algorithm) menu. The entries each invoke one of the compositional algorithms described in section 4.3. The performance parameters for these invocations is compiled into the program; relatively inflexible, this menu is primarily provided as a means of testing and demonstrating the basic texture implemented by each algorithm.

7.5.5 Other Commands

The push buttons in the upper left corner of the interface, to the right of the state slider, each execute some task every time they are clicked by the user. The buttons *Single* and *Free* have already been described in the state selection section. The button marked *Hung* is a MIDI panic button, used to terminate hung notes. Anyone who has used MIDI will be familiar with the phenomenon of the stuck note, which occurs when a Note On command somehow gets sent out without the corresponding Note Off command. When that happens, the sound simply goes on interminably. The *Hung* button on the interface will silence any hung notes, by the rather direct device of sending out a Note Off command to every note number on every channel. The rudely named button *ShutUp* will also terminate any stuck notes, with the further effect of preventing the program from producing any more sound. Sound will again issue from the program when the button *Clear* is clicked; *Clear* will also disconnect all level1 and level2 listener and player agents.

To the right of the pushbuttons are three text controllers, marked *Record*, *Channel*, and *Echo*. These will cycle through several positions, whenever the user clicks on the title of the operation. For example, clicking on the *Record* title will alternate between the *On* and *Off* positions. With recording turned on, the program will send the output of the composition section to both the MIDI interface, and a disk file following the standard MIDI file format. Such a disk file permits exporting the music produced to other applications able to handle the standard file format, such as sequencers, notation packages, etc.

The *Channel* title cycles between four possibilities, marked *Kbd*, *Ivl*, *Nobody*, and *Unknown*. This refers to the MIDI source which is being tracked by the listener. The *Kbd* and *Ivl* sources will typically be assigned to different MIDI channels, and *Nobody* as a source prevents input from any source from reaching the listener, so that the program will be thrown into composition by introspection. Which channels correspond to the keyboard or the Ivl pitch-to-MIDI converter is established in the *cypher.voices* configuration file, the same file used to allocate synthesizer resources to the *Bank* and *Timbre* radio buttons. The value of the *Channel* controller can be changed from the program itself, in which case the interface will display the current value of the channel input variable. When the program sets the listener to track a channel which does not correspond to the settings for either the keyboard or pitch-to-MIDI converter (Ivl), the display will read *Unknown*.

The last text controller, *Echo*, cycles again through the possibilities On and Off. When *Echo* is on, the MIDI data arriving from the device indicated by the channel input controller will be copied and sent out to the synthesizers selected by the *Bank* and *Timbre* buttons. This provides a convenient way to work on the *cypher.voices* file, trying out different voice combinations by directly playing them from the keyboard, for instance, rather than hearing them only as an expression of program output.

Chapter 8

Music Performances

During the development of *Cypher*, numerous musical works were performed which tested and extended the algorithms being implemented. Though they provided a powerful motivation and rigorous environment for experimenting with the program, they are not themselves experiments, or, as a consequence, to be considered "experimental music." The compositions and improvisations described below are complete, performed, functional pieces of music, which should be considered by musical standards, beyond their purely technical advances. The exposition below will concentrate on the features of *Cypher* explored by each piece; such a style of presentation, however, is not meant to obscure their essence as musical compositions.

These works fell into two broad categories : largely composed pieces, and completely improvised performances. There were four composed works (which themselves all include some forms of improvisation), using human ensembles ranging from one to eight players. Musicians were constantly improvising with the program, usually on an informal basis, but on three occasions *Cypher* contributed to a more or less completely improvised performance in concert. This chapter describes musical works performed with *Cypher*, focusing on software developments which made the new kinds of interaction featured in each work possible.

8.1 Composed Works

The category of composed works comprises those pieces whose performance relies on a notated score. In these compositions, the human players are asked to perform at various times through either of two means of musical expression: interpretation of a written part, or improvisation. Interpretation comes into play during the performance of completely notated sections, in which case the players behave as the interpreters of Western concert music normally do: reading the music to be performed from the page, and rendering it with their own, trained musical judgement. In other sections, the players are given the freedom to improvise the music they perform, in keeping with

the spirit of the composition. Usually, some constraints or indications of basic material are given to the players in the improvised sections; ideally, however, the improvisations are rehearsed and honed through the collective efforts of composer, performer, and computer.

8.1.1 Flood Gate

The first composed work written with *Cypher* was *Flood Gate*, a trio for violin, piano, and computer. As in all of the notated compositions, the human performers are presented with a score which includes conventionally notated music, and some partially or completely improvisatory sections. The software development for this work concentrated on a number of the system's fundamental performance tools: score orientation, studio to stage compositional refinement, and algorithmic responses to improvisation.

In the completely notated sections, the output of the computer part is virtually identical from performance to performance, since the same material is being played into the same highly deterministic processes[6]. In the improvised sections, the output of the computer part is also a transformation of the human performers' input: the transformations used are a function of the features extracted from the music played to the computer. Each performance of the improvised sections is different; one goal of the composition was to marry composed and improvised elements in the same piece, using similar algorithmic techniques. Score orientation guided *Cypher* from one part of the composition to the next; sixty-one state changes are performed at precise moments in the course of a ten-minute piece, with no operator intervention.

Most of the processing used in this piece took place on level 1 of both the listener and player. This being the first essay, the most immediate forms of

---

[6]There are some random numbers in these processes, but they are used in tightly constrained ways. For example, the ornamentation filter will add two pitches in close proximity to the input pitch. Though the ornamental pitches are chosen at random within a small ambitus, the effect of a small flourish around a target pitch is the perceived result; in other words, the use of random numbers is so tightly restricted that the relation of input to output is virtually deterministic.

analysis and reaction were used. The work succeeds as an example of interaction on a featural level, but also points out the advantage of being able to guide processing on higher musical levels as well: phrase-length effects such as crescendo and accelerando are largely missing from the computer output. Following is a page from the score, illustrating the material with which the pianist was presented for her improvisation.

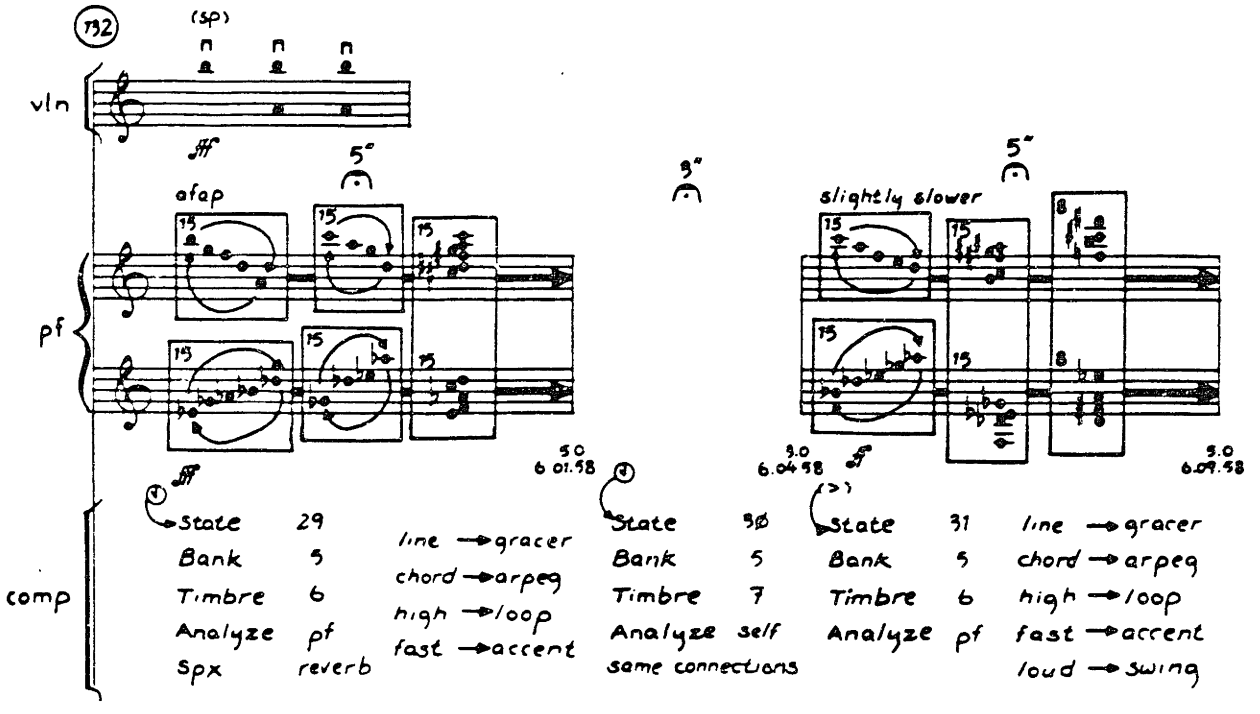


Fig. 45

8.1.2 Banff Sketches

The next composition to use *Cypher* was a work for solo piano and computer, called *Banff Sketches*. This piece was, appropriately enough, begun during a residency at the Banff Centre in Alberta, Canada, during the summer of 1989, in collaboration with the pianist Barbara Pritchard. Subsequent performances at the SIG-CHI '90 conference, a *Collage* concert of Media Laboratory works at Symphony Hall in Boston, and at the MIT Media Laboratory concert performed at the completion of this thesis provided opportunities to refine and expand the composition.

One of the main extensions added to the program for *Banff Sketches* was *alternation*. This is a mode designed to emulate the well-known compositional and performance practice of taking turns to elaborate some common material; in jazz a typical application of the idea is "trading eights", where two performers will alternate improvisations of eight bars each, commenting and expanding on ideas introduced by the other. For such an idea to work with *Cypher*, the program had to know when to store material from the human player, and when to compose an elaboration of that material in an unaccompanied presentation.

In *Banff Sketches*, alternation patterns were stored as part of the score orientation cue list, which informed the program whose turn it was to perform. During the human performance, *Cypher* stored what was being played, unanalyzed. When the program was due to treat the same material, it was sent through to the listener and transformed through message/method connections in the same way that composition is usually performed simultaneously.

Further, *Banff Sketches* provided the most extensive opportunity for improvisation of any of the principally composed works. In contrast to the organization of *Flood Gate*, improvisations are scattered throughout the composition, as commentaries and extensions to most of the musical ideas presented in the work. Computer solos often follow human/computer improvisatory collaborations, continuing to develop the material just played. *Cypher*'s performance in these sections included the first extensive use of the

composition critic; modifications to the behavior of the program were not only a function of state changes coupled to score orientation, as in the case of *Flood Gate* or *Sun and Ice*, but also were effected by meta-level rules incorporated in the critic watching the composition section's output.

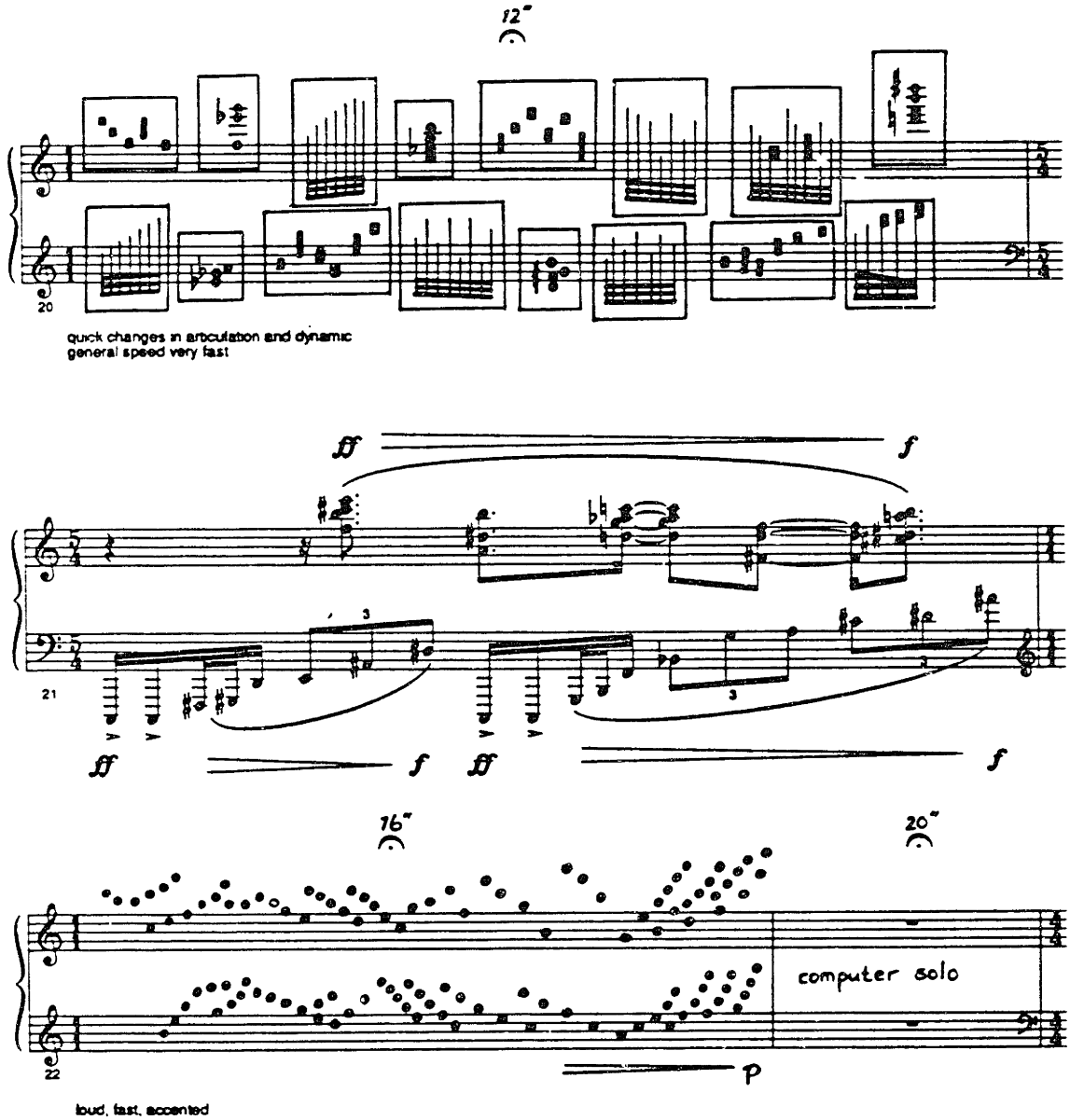


Fig. 46

An example of these ideas can be seen on page 12 of the score; the first measure shown is a graphic representation of improvisation possibilities for the performer. The human performer may rely on these indications almost literally; a possible mapping of the graphic symbols to performance is provided with the score. If the player is more comfortable with improvisation, the symbols may be taken as little more than a suggestion. In all of the performances to date, the composer has had the luxury of working together with the performer to arrive at appropriate and engaging kinds of improvisation for each section.

8.1.3 Sun and Ice

Sun and Ice is a composition for four human performers and *Cypher*. The human ensemble performs on flute, oboe, clarinet, and a MIDI percussion controller. The percussion controller is directly connected to the program; the three wind players are tracked through a pitch-to-MIDI converter which takes an acoustic signal (all three wind instruments are played into microphones) and returns a MIDI representation of the pitches being played. Only one of the human players is ever being tracked by *Cypher* at a time; therefore, an important formal decision in the composition of the piece was mapping out which human performer would be interacting with the program when.

Sun and Ice makes the least use of improvisation of any of the composed works. Structurally, the piece falls out into four big sections; at the end of the second of these, there is an extended improvisation section for the three wind players, followed by a computer solo. Here again we see the principle of a collaborative improvisation coupled with an elaboration of the improvised material by *Cypher*.

The technique taken furthest in *Sun and Ice* was that of using the compositional algorithms as seed material for introspective improvisation by the program.

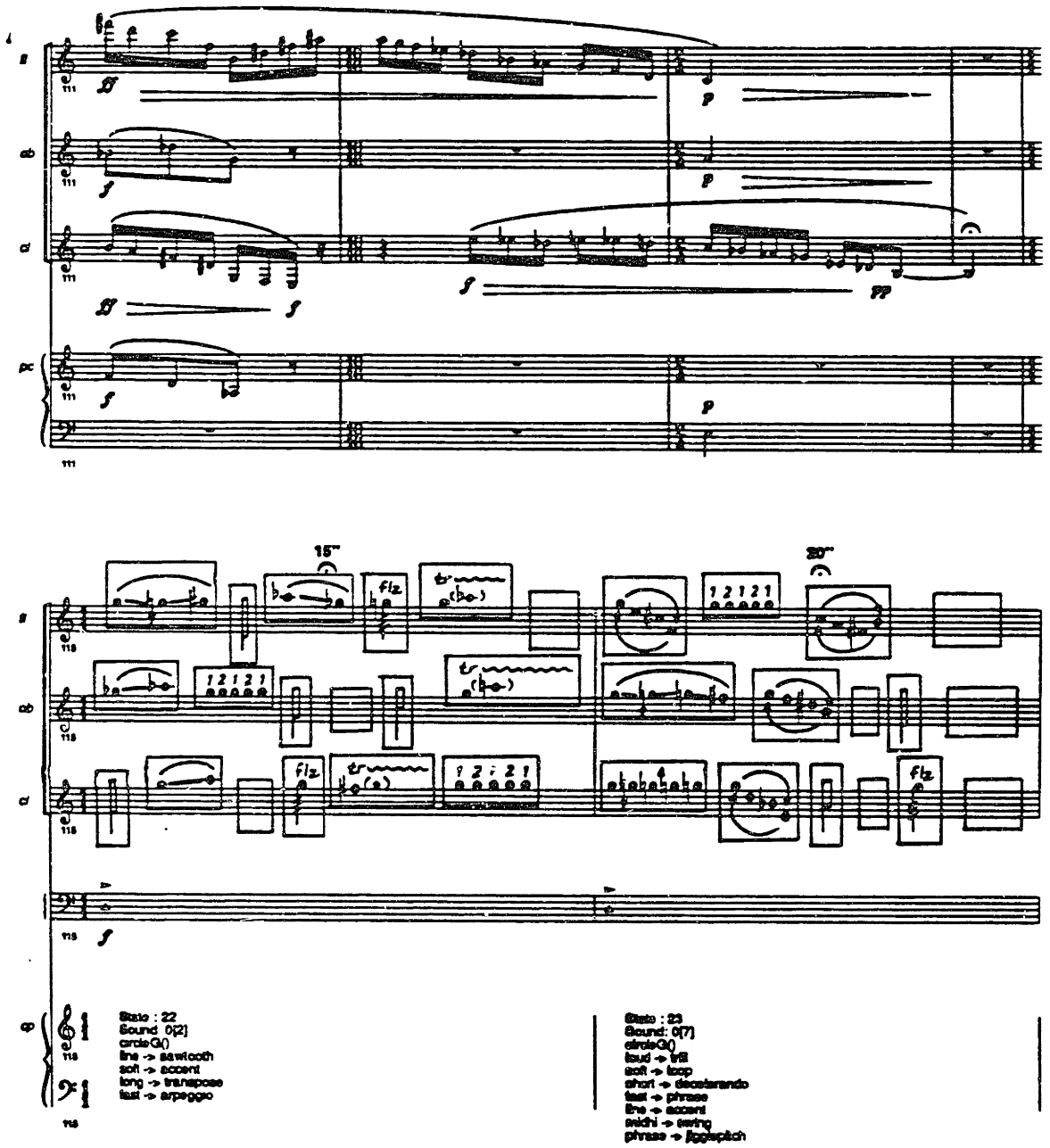
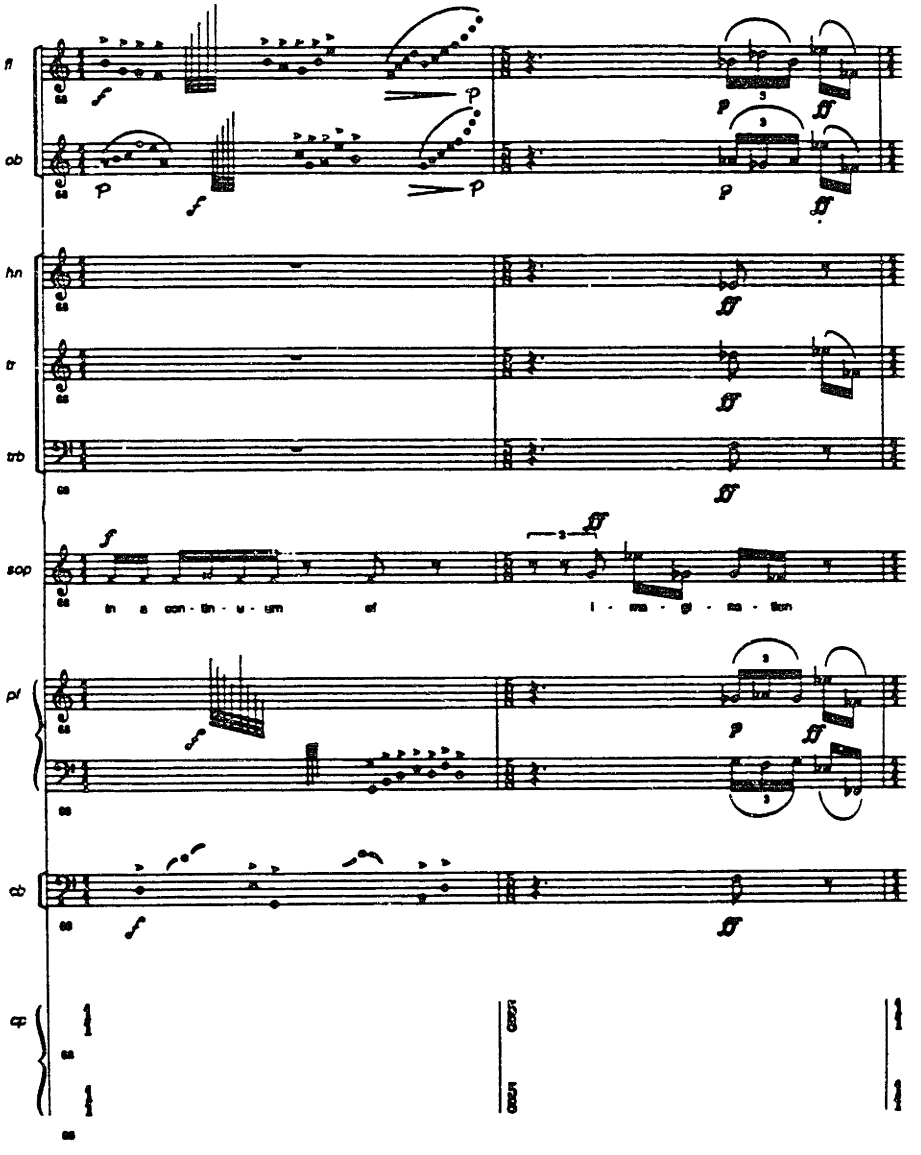

15"
20"
fl2
tr
1 2 1 2 1
f
ff
p
pp
State : 22
Sound: 0[2]
circleG()
line -> sawtooth
soft -> accent
long -> transpose
fast -> arpeggio
State : 23
Sound: 0[7]
circleG()
loud -> trill
soft -> loop
short -> decelerando
fast -> phrase
line -> accent
swidhi -> swing
phrase -> jiggleplich

Fig. 47

8.1.4 Rant

The most extensive composed work was *Rant*, a piece for flute, oboe, horn, trumpet, trombone, piano, double bass, soprano voice, and *Cypher*, based on a poem by Diane Di Prima. This piece was premiered on April 19, 1991, with soloist Jane Manning and as conductor Tod Machover. The piece contrasted several sections for parts of the ensemble against other material presented by the group as a whole.



One of the most challenging compositional problems presented by *Cypher* is the question of how to integrate it with a large ensemble. Full musical textures tend to integrate the contributions of many instruments into unified gestures or sets of voices. Adding an active computer complement to such a context can easily produce a muddled, unfocused result. In *Rant*, the solution adopted to the ensemble integration problem was to put the activity of the computer part in an inverse relation with the performance of the human ensemble. When the rest of the ensemble was performing together, *Cypher* adopted a strictly subsidiary role. In the numerous sections for smaller forces, the program came to the fore. In particular, two parts of the piece were written for soprano and computer alone, highlighting the interaction between the vocal material and *Cypher*'s transformations.

8.2 Improvised Works

The following works used *Cypher* as a performer in a completely improvised setting: little if anything was agreed upon about the nature of the performance in advance, and the program had to contribute to the texture through its analysis capabilities, and relations to composition methods which were either specified live, using the interface, or generated by the critic-driven composition rules.

8.2.1 Concerto Grosso #2

Concerto Grosso #2 was the public debut of *Cypher*. This performance, part of the Hyperinstruments concert given at the Media Laboratory in June of 1988, combined the improvisations of Richard Teitelbaum on piano, his improvisational software *The Digital Piano*, George Lewis on trombone, Robert Dick on flute, and *Cypher*. Routing of MIDI signals between the acoustic instruments (which were initially sent through pitch-to-MIDI converters) and computers was done onstage by Richard Teitelbaum. A subsequent, reduced performance was given by Richard Teitelbaum and Robert Rowe at the Concertgebouw in Amsterdam in the fall of 1988.

This earliest form of the program used the featurespace analysis on incoming events as an index into a table of possible responses, which included the

performance of sequences, and the transformation of incoming material. The image of the relation between featurespace and response at that time was that each input event would occupy one point in the space (as is still the case). Found in that point was a routine which was to be executed whenever an event was placed there. As the featurespace grew, and the vast majority of points came to be filled with the current techniques of calling transformations based on the featural analyses, this indexing mechanism was eventually abandoned.

8.2.2 Universe III

Universe III was a completely improvised musical performance given by human and machine improvisers. The piece was played at the Banff Centre in the summer of 1989. The human players were Steve Coleman on saxophone and synthophone, and Muhal Richard Abrams on MIDI piano. The machine players were Coleman's improvisation program, and *Cypher*. Configuration of the MIDI streams was done onstage, so that the input to *Cypher* changed continually between the other human and machine performers, and itself.

The most intriguing result of this performance and the rehearsals leading up to it was the speed with which the jazz players were able to pick up on the transformations tied to features of their playing, and trigger them at will. Connections between the *Cypher* listener and player were made onstage: a human operator (myself) performed state changes and other major modifications to program behavior.

8.2.3 Coleman Collaboration

On the "Computer Jazz" concert of April 19, 1991, another collaboration combined the live synthophone playing of Steve Coleman with additional voices supplied by his own improvisation software, and *Cypher*. The synthophone's MIDI output was connected to three computers: one running Coleman's software, a second running *Cypher*, and a third playing alto saxophone samples. Since the Banff Centre performance, Coleman's software improviser has been extended to track live playing and find harmonic

structure in it. The harmonies found are then used to direct the pitch selection of the melodic improvisation part.

In this performance, *Cypher* was run to a large extent by an internal software critic. The normal production rules, adjusting the output of the composition section just before performance, remained in force. Additional rules examined the analysis of Coleman's playing and made or broke connections maintained in the connections manager between listener reports and compositional methods. Further, extensive use was made of the harmonic and rhythmic adjustment transformations, tracking the behavior of the synthophone performance to pick up the harmonic context and beat period, then aligning the output of Cypher's composition section to accord with it. The performance of the beat alignment transformation was particularly noticeable; once the beat tracker locked onto Coleman's pulse, the composition section achieved an effective rhythmic coordination with the live playing.

Chapter 9

Related Work

Several researchers have implemented interactive music systems exhibiting some of the same input sensitivity and real-time response of the work described above. Though none duplicate the effort that produced *Cypher*, some are clear precursors, and their concerns, successes, and problems, have helped shape the development of the work reported. In what follows, I will first review several important interactive systems individually, then sketch out a framework in terms of which they might be evaluated, and use it to draw out common features and solutions. In this review, I will outline ways to assess the contribution of interactive computer systems in terms of the musical problems they address. Now we are able to hear the results of compositional and analytical processes: next we need ways to evaluate and profit from them.

9.1 Evaluation

From the *Iliac Suite*, G.M. Koenig's *PR1*, and Xenakis' early formalized music programs to the present day, composers have employed computers to enhance and extend their own compositional practices [Loy 1989]. A term which unites many of these projects is *algorithmic composition*: the use of a computable method for the generation of music. Strictly speaking, it is inaccurate to say that these programs are concerned with modeling human compositional techniques: usually, they are used to work out some procedure with greater speed, more accuracy, and less "cheating" than their human creators would be able to muster. In particular, composers use compositional algorithms to develop high level methods, procedures which operate on a control plane governing the generation of notes and durations, without forcing the composer to consider each event individually.

To assess the contribution of such efforts, then, we may attempt the following method: determine the musical intention of the composer, done, if at all, by listening to the music the program produced; look for the algorithms and control variables used in the program to realize those intentions; and analyze

the effectiveness, flexibility, and power of the technique in terms of the relation between the musical intent, the algorithm used, and the sounding, perceived result.

There are problems with this, naturally: first of all, many of the people making the systems aren't talking, either about the music or the software. Assessment often becomes a matter of experienced guesswork, led by the ear and observations of the way the program is used onstage. The fact, mentioned in Chapter 1, that most work is done by the people making a particular piece, rather than through any generalized effort or platform, means that there is little common ground or motivation for discussing at length the relation between system and music.

This observation should not be taken as an accusation of deliberate obfuscation on anyone's part; the dialog problem sketched here is a natural outgrowth of a situation where the people doing the work are more interested in making new pieces than in demonstrating the techniques of the old ones. Even given these obstacles, a number of efforts have been around long enough, and seen in enough manifestations, that we may attempt a discussion of them: we can, even if fitfully, learn from each other.

9.2 Classification

Let us introduce some metrics to classify, and thereby begin to evaluate, interactive music systems. The first dimension distinguishes systems which are *score-driven* from those which are *performance-driven*.

• Score-driven programs use sequences of predetermined events (stored music fragments) which are likely to be organized using the traditional categories of beat, meter, and tempo. Such categories allow the composer to preserve and extend familiar ways of thinking about temporal flow, e.g. specifying some events to occur on the downbeat of the next measure, or at the end of every fourth bar.

• Performance-driven programs do not include sequences. Their independence from stored material also renders them unable to profit from prior structuring of material, such as into beats and measures. Often this means that performance-driven programs do not employ traditional metric categories for temporal organization, but tend to use as parameters perceptual measures, such as density and regularity, to generate event offsets. If the program is able to measure metric structure during performance, these means may be extended to include more traditional concepts.

These two possibilities are not strictly exclusive; primarily performance-driven systems may well use some kind of score, for some fragment of fully notated music, or as a record of kinds of generation to be associated with specific parts of a human performance. Similarly, score-driven programs may also track aspects of the live performance, and use these to direct the rendition of stored material. Rather than a strict division, score- and performance-driven programs are more an orientation, which is marked most clearly by the presence or absence of stored sequences in the program's output.

Another distinction classes algorithmic approaches as *transformative, generative,* or *sequenced.*

• Transformative algorithms take some pre-existing musical material and apply transformations to it to produce variants; according to the technique, these variants may or may not be recognizably related to the original. For transformative algorithms, the source material is complete musical input and a list of transformations to be applied to it.

• For generative algorithms, on the other hand, what source material there is will be elementary or fragmentary - for instance, stored scales or duration sets. Generative methods will then use rules to produce complete musical output from the stored fundamental material, taking pitch structures from basic scalar patterns according to random distributions, for example, or applying serial procedures to sets of duration values.

• Sequenced techniques use pre-recorded music fragments in response to some real-time input. Some aspects of these fragments may be varied in performance, such as the tempo of playback, dynamic shape, slight rhythmic variations, etc.

Finally, we can distinguish between the *instrument* and *performer* paradigms.

• Systems following an instrument paradigm are concerned with constructing a kind of musical instrument: performance gestures from a player are used to guide the instrument in producing musical output. Imagining such a system being played by a single performer, the musical result would be experienced as a solo.

• Systems following a performer paradigm try to construct an artificial player, a musical presence with a personality and behavior of its own, though it may vary in the degree to which it follows the lead of a human partner. A performer paradigm system played by a single human would produce an output more like a duet.

9.3 Score Following

Score followers are programs which match the performance of a human player against an internal representation of the music that player will perform. The matching process advances a pointer through the internal representation, and a comparison of the performed temporal offsets between events with the stored offsets produces an estimate of the tempo of the live performance on a moment-to-moment basis. The tempo estimate is then used to drive a synthetic realization of an accompanimental score [Vercoe84; Dannenberg89].

Score followers are the first interactive systems to be based on an anticipation of the behavior of a human performer: the derivation of tempo is in fact a prediction of the future spacing of events in a performance based on an analysis of the immediate past. As discussed in Chapter 6, score followers

implement the most advanced pattern matching techniques used in interactive systems in their comparison of the live performance with the stored score. The match must be reasonably fault-tolerant in the face of wrong notes, skewed tempi, or otherwise "imperfect' performances rendered by the player. Vercoe's system is able to optimize response to such performance deviations by remembering the interpretation of a particular player from rehearsal. Successive iterations of rehearsal between machine and human results in a stored soloist's score which matches more precisely a particular player's interpretation of the piece.

The musical implications of score following in its simplest form are modest: such applications amount to tempo-sensitive music-minus-one machines, a technical improvement of the recordings of concerti with the solo part omitted. The potential range of application of these techniques, however, is much broader. Real-time pattern matching will be a critical component of any interactive system incorporating a memory of progressions; many of the same matching techniques developed for score following can be used to look for repetitions of known progressions in new input. Further, the fault-tolerance of score following can be expanded to find variants of known progressions within some bounds. Successful matches, or partial matches, can then be used to anticipate the continuation, extending the most remarkable facet of score following – the ability to anticipate the nature of imminent performance.

9.4 Hyperinstruments

The *hyperinstruments* of Machover and Chung view the interactive system as an instrument to be learned and played by a virtuoso performer [Machover & Chung 1989]. Their work has resulted in powerful and varied levels of control over complex musical material through the extension of traditional instrumental technique. For example, several keyboard-based applications make use of pianistic playing technique, but reinterpret those gestures in a manner appropriate to the nature of the material being played. The arpeggiation instrument animates chords performed on a keyboard with quickly changing rhythmic sequences applied to an arpeggiation of the notes of the chord. The performer controls movement from one chord to the next, and, in an extension of normal technique, can affect the timbral presentation

of the arpeggios by applying and releasing pressure on the held notes of the chord. Moreover, the program is continually searching through a database of chords to find one which matches what is being played on the keyboard; successful matches are used to move from one rhythmic sequence to the next.

Hyperinstruments are used in pieces of composed music, where an important structural element is the changing pattern of relationships between acoustic and computer instruments. Hyperinstruments enable the performance of complex musical material in a way that is sensitive to the gesture of the human player. The most important software design strategy for the realization of these musical goals involves the construction of generic computational objects, such as standard score representations, or time grid manipulations, with which are built the implementations needed for different kinds of instrumental interaction throughout the composition [Machover & Chung 1989]. Though the musical demands of different pieces, or different sections of a single piece, may vary, adaptation of the generic objects enables consistent, reliable realizations of those demands.

9.5 Automatic Improvisation

Automatic improvisation systems generate new musical material in situations where little or nothing is known about the nature of the human performance in advance. In a system developed by George Lewis, no sequences or other precomposed fragments are involved, and all output is the product of operations on stored elemental material, such as scales and durations. Interaction arises from the system's use of information arriving at the program from outside sources, such as a pitch-to-MIDI converter tracking Lewis' trombone.

A listening section observes and parses the MIDI input, and posts the results of its observations in a place accessible to the routines in the generation section. The generation routines, using the scales and other raw material stored in memory, have access to, but may or may not make use of, the publicly broadcast output of the listening section [Lewis 1989]. Probabilities play a large role in the generation section; various routines always are

available to contribute to the calculations, but are only invoked some percentage of the time - a probability which is set by the composer, or which changes according to analysis of the input.

The probability of durational change, for example, is related to an ongoing rhythmic complexity measure, so as to encourage the performance of uneven, but easily followed sequences of durations. Lewis has a technique for beat following which depends on the detection of a "sub-tempo", some short duration which, when added to itself some integer number of times, will account for most of the actual durations being heard. In other words, the Lewis algorithm is additive, looking to find large durations from combinations of short ones, rather than looking for a tempo of longer values and subdividing it to account for the quick notes.

George Lewis' software performers are designed to play in the context of improvised music. The intent is to build a separate, recognizable personality which participates in the musical discourse on an equal footing with the human players. The program has its own behavior, which is sometimes influenced by the performance of the humans. The system's success in realizing Lewis' musical goals, then, follows from these implementation strategies: the generative nature of the algorithm ensures that the program has its own harmonic and rhythmic style, since these are part of the program, and not adopted from the input. Further, the stylistic elements recognized by the listening section are made available to the generation routines in a way which elicits responsiveness, but not subordination, from the artificial performer.

9.6 Input Transformation

Richard Teitelbaum's *Digital Piano* collection, controlled by Teitelbaum and Mark Bernard's *Patch Control Language* [Teitelbaum84], includes a MIDI input keyboard, a computer equipped with several real-time physical controllers, such as sliders and buttons, and some number of output devices, usually including one or more solenoid-driven acoustic pianos. Using the patch control language, Teitelbaum is able to route musical data through a

combination of transformations, including delays, transpositions, and repetitions. Before performance, Teitelbaum specifies which transformation modules he intends to use, and their interconnections. During performance, material he plays on the input piano forms the initial signal for the transformations. Further, he is able to manipulate the sliders to change the modules' parameter values, and use buttons or an alphanumeric keyboard to break or establish routing between modules.

In this case, the musical intelligence employed during performance is mostly Teitelbaum's: the computer system is a wonderful generator of complex, tightly knit musical worlds, but the decision to change the system from one configuration to another is the performer's. The computer does not decide on the basis of any input analysis to change its behavior; rather, the composer/performer sitting at the controls actively determines which kinds of processing will be best suited to the material he is playing.

Richard Teitelbaum's software is designed to function in an improvisational context. The musical intent is to establish a context of variations on the improvised input, which will reflect and extend the original gesture. Expression is transmitted through the system by using a human performer's input as the basis for program output; subtleties of timing and dynamic are preserved, unless they are expressly overridden by some transformation. The technique of chaining together relatively simple transformation modules allows the composer to produce variations along a continuum of complexity, where the number and type of transformations chained together directly affect the degree and kind of complexity in the output. Similarly, the output is more or less recognizably related to the input as a function of the number of processes applied to it. The system tends to resemble a musical instrument more than a separate player, since the decision to change its behavior is made by the composer during performance, rather than by the program itself.

9.7 Artificial Performer

Jean-Claude Risset's *Etudes* were written for an interactive system programmed with Miller Puckette and David Zicarelli's *Max* environment [Puckette88]. Using *Max* , the composer specifies a flow of MIDI or other

numerical data among various operations such as addition, scaling, transpositions, delays, etc. Operations are represented by graphic objects, which can be connected and moved so as to easily control data flow between them, inspect temporary or final results along any point in the path, and introduce or modify real-time processing. Further, patches can be constructed hierarchically: once a working configuration of objects for some process has been found, it is saved to a patch, which can be included as a whole in other patches.

Risset's *Etudes* are written for the Yamaha Disklavier, played as if with four hands. That is to say that while a human performer plays two hands' worth of the piece, MIDI data from that performance is transmitted to a computer running *Max*, which applies various operations to the human-performed data, and then sends out MIDI commands to play two more hands' worth on the same piano. The variety of techniques used in the *Etudes* are a tribute to the flexibility of *Max* – more than an interactive system, *Max* is a graphic programming language for designing interactive systems. All of the *Etudes* are score-driven, and follow a player paradigm. Both sequenced and transformative generation techniques are used: *Double* plays sequences in response to notes found in the score, and *Fractals* adds several notes in a quasi-octave relationship to each note played.

As with the hyperinstruments environment, the most important relationship between the computer program and musical application to be noted is that many specific compositional demands can be met through a simple reconfiguration of existing software objects. The mapping of musical thought to computer realization is accomplished through an application of the tools at hand. The *Etudes* are composed, notated pieces, whose musical demands were met and shaped by the collection of generic operations *Max* provides.

Chapter 10

Summary

> *Trouble would begin, however, if mechanical music were to flood the world to the detriment of live music, just as manufactured products have done to the detriment of handicraft. I conclude my essay with this supplication: May God protect our offspring from this plague!* – Bela Bartok [Bartok37 p. 298]

The computer program described here, *Cypher*, coordinates the output of many small, independently functioning agents. Analytically, these agents combine to make larger agencies which perform complex tasks, such as chord identification, key identification, beat tracking, and phrase grouping. Compositionally, collections of agents can be engaged in the generation of new musical output, which is in turn modified through the preferences of a compositional critic, a separate agency devoted to the evaluation of musical structures and their evolution in time.

The structure of such agencies, and the representation of the objects on which they operate, embody a multiplicity of perspectives on the ongoing musical fabric. According to the task at hand, events may be regarded as articulating a progression through time, or as members of hierarchically arranged groups, or as points in a featurespace. Processes manipulating these events may operate in parallel, or in groups executed in stages, or as cooperating agencies sending partial or qualified information across a two-way communication link.

This thesis demonstrates that concentration on the communication and coordination of many small agents allows the concurrent analysis and composition of sophisticated musical structures in real time. Further, pattern processing of MIDI input can recognize and match sequences and some simple variants, allowing the program to learn frequently repeated strings. When a match has been made or is ongoing, messages to the composition section allow the program to complete a partial match, or use the found pattern as seed material for some collection of transformations.

This software design is linked to a perspective of compositional and performance practice which encourages continual refinement of musical ideas from general, high-level plans to a specific sequence of changing states realizing a particular set of compositional goals in a performance context. The graphic interface, state save and restore mechanism, and score orientation techniques are all expressions of such a perspective. The fruits of this research have been repeatedly demonstrated in live performances, of either completely notated works, completely improvised situations, or, most commonly, in a new kind of formal organization which combines interpretation with improvisation.

The changes in my own musical thought during the development of *Cypher* have also been related to the new formal possibilities the program provides. I have long been interested in developing ways to involve the player in shaping important aspects of the composition during performance. The improvisational segments of pieces written with *Cypher* show an organic relation to the rest of the composition, because, I believe, the program knits together composed and improvised material in a convincing way. Reactions to composed music and improvisations are organized in the same way; the improvisational sections will show great variation from performance to performance, while the composed parts stay relatively constant. However, sharing gestures between the two styles of performance, and weaving improvisational elaboration of composed material into an ongoing presentation of composed music, (as in *Banff Sketches*, or *Flood Gate*), seems to make a coherent whole out of performances that might otherwise fall into irreconcilable pieces.

In an article in *The New York Times* of May 13, 1990, titled "The Midi Menace: Machine Perfection is Far From Perfect" Jon Pareles noted with distress the burgeoning use of sequenced MIDI material as a substitute for human players in the performance of pop music. He writes: "If I wanted flawlessness, I'd stay home with the album. The spontaneity, uncertainty and ensemble coordination that automation eliminates are exactly what I go to concerts to see; the risk brings the suspense, and the sense of triumph, to live pop." [Pareles90]

Similarly, I have often heard professional musicians complain of the development of computer music that soon all the performance work would go to machines. At first, I took such complaints as evidence of a lack of familiarity with the field: it seemed to me there was little danger of machines taking over the concert stage as long as they remain such remarkably poor musicians. And yet, as Pareles notes, machines are assuming an ever-increasing role in the performance of music. Because of the nature of the machine's participation, such occasions come to resemble less a live performance, than the public audition of a tape recording.

In developing my own computer musician, I attempt to make human participation a vital and natural element of the performance situation. Not in the first place because I am concerned about putting performers out of work: but rather because I believe that if the numbers of humans actively, physically making music declines, the climate for and quality of music-making in general will continue to deteriorate. Pareles concludes by saying: "Perhaps the best we can hope for is that someone will come up with a way to program in some rough edges, too." I hope we can do better than that: develop computer musicians that do not just play back music *for* people, but become increasingly adept at making new and engaging music *with* people, at all levels of technical proficiency.

References

[Bartok37] Bartok, B."Mechanical Music" (1937) in *Bela Bartok Essays* Selected and Edited by Benjamin Suchoff London : Faber & Faber 1976 pp. 289-298

[Berry76] Berry, W., *Structural Functions in Music* New York: Dover Publications, Inc. 1976

[Bharucha87] Bharucha, J. J., "Music Cognition and Perceptual Facilitation: A Connectionist Framework" *Music Perception* 5 1987 pp. 1-30

[Bharucha & Todd89] Bharucha, J. J. & Todd, P. M., "Modeling the Perception of Tonal Structure with Neural Nets" *Computer Music Journal* 13:4 1989 pp. 44-53

[Bloch&Dannenberg85] Bloch, J. J., & Dannenberg, R. B., "Real-Time Computer Accompaniment of Keyboard Performances"*Proceedings of the ICMC 1985* pp. 279-289

[Boynton87] Boynton, L., "Scheduling as Applied to Musical Processing" M.I.T. Media Laboratory report November, 1987

[Brün85] "Interview with Herbert Brün" Peter Hamlin with Curtis Roads in *Composers and the Computer* Curtis Roads, ed. Los Altos, CA: William Kaufmann, Inc. pp. 1-15

[Chung89] Chung, J., "An Agency for the Perception of Musical Beats, or, If I Only Had a Foot..." M.I.T. Media Laboratory report June, 1989

[Clarke87] Clarke, E.F., "Categorical Rhythm Perception: An Ecological Perspective" in *Action and Perception in Rhythm and Music* Alf Gabrielsson, ed. Stockholm: Publications issued by the Royal Swedish Academy of Music 1987 pp. 19-34

[Cook87] Cook, N., *A Guide to Musical Analysis* New York : George Braziller 1987

[Cope90] Cope, D. "Pattern Matching as an Engine for the Computer Simulation of Musical Style" *Proceedings of the ICMC 1990* pp. 288-291

[Dannenberg & Mont-Reynaud87] Dannenberg, R. B., & Mont-Reynaud, B., "Following an Improvisation in Real Time" *Proceedings of the ICMC 1987* pp. 241-247

[Dannenberg89] Dannenberg, R. B., "Real-Time Scheduling and Computer Accompaniment" in *Current Directions in Computer Music Research* Mathews, Max V. and Pierce, John R., eds. Cambridge: MIT Press 1989 pp. 223-261

[Handel90] Handel, S. *Listening : An Introduction to the Perception of Auditory Events* Cambridge: MIT Press 1990

[Jaffe85] Jaffe, D., "Ensemble Timing in Computer Music"*Computer Music Journal* 9:4 1985 pp. 38-48

[Krumhansl90] Krumhansl, Carol L. "Melodic structure: Theoretical and perceptual aspects" International Wenner-Gren Symposium: Music, Language, Speech, and Brain Stockholm, Sweden 1990

[Lerdahl88] Lerdahl, F. "Cognitive constraints on compositional systems" in *Generative Processes in Music: The Psychology of Performance, Improvisation, and Composition* John Sloboda, ed. Oxford: Clarendon Press 1988 pp. 231-259

[Lerdahl&Jackendoff83] Lerdahl, F. and Jackendoff, R., *A Generative Theory of Tonal Music* Cambridge: The MIT Press 1983

[Levitt81] Levitt, D. "A Melody Description System for Jazz Improvisation" Master of Science Thesis MIT 1981

[Lewis85] "Improvisation with George Lewis" interview by Curtis Roads in *Composers and the Computer* Curtis Roads, ed. Los Altos, CA: William Kaufmann, Inc. pp. 75-88

[Loy85] Loy, G., "Musicians Make a Standard: The MIDI Phenomenon" *Computer Music Journal* 9:4 1985

[Loy89] Loy, G., "Composing with Computers - a Survey of Some Compositional Formalisms and Music Programming Languages" in *Current Directions in Computer Music Research* Mathews, Max V. and Pierce, John R., eds. Cambridge: MIT Press 1989 pp. 291-396

[Machover&Chung89] Machover, T. and Chung, J. "Hyperinstruments: Musically Intelligent and Interactive Performance and Creativity Systems" *Proceedings of the ICMC 1989* pp. 186-190

[McAdams87] McAdams, S., (ed.) *Music and Psychology : A Mutual Regard* Contemporary Music Review, Volume 2 Part 1 London : Harwood Academic Publishers 1987

[Meyer56] Meyer, L. B., *Emotion and Meaning in Music* Chicago: University of Chicago Press 1956

[Minsky75] Minsky, M., "A Framework for Representing Knowledge" (1975) *Readings in Knowledge Representation* Ronald J. Brachman and Hector J. Levesque, eds. Los Altos: Morgan Kaufmann Publishers, Inc. 1985 pp. 245-262

[Minsky81] Minsky, M., "Music, Mind, and Meaning" (1981)*The Music Machine* Curtis Roads, ed. Cambridge: MIT Press 1989 pp. 639-656

[Minsky86] Minsky, M., *The Society of Mind* New York: Simon and Schuster 1986

[Moore88] Moore, F.R., "The Dysfunctions of MIDI" *Computer Music Journal* 12:1 1988 pp. 19-28

[Moore90] Moore, F.R., *Elements of Computer Music* Englewood Cliffs: Prentice Hall 1990

[Narmour77] Narmour, E. *Beyond Schenkerism: The Need for Alternatives in Music Analysis* Chicago: The University of Chicago Press 1977

[Narmour84] Narmour, E. "Some Major Theoretical Problems Concerning the Concept of Hierarchy in the Analysis of Tonal Music" *Music Perception* 1:2 Winter 1983-84 pp. 129-199

[Palmer&Krumhansl87] Palmer, C., & Krumhansl, C. L. "Independent temporal and pitch structures in determination of musical phrases" *Journal of Experimental Psychology: Human Perception & Performance* 13 pp. 116-126

[Palmer88] Palmer, C. "Timing in Skilled Music Performance" Ph.D. dissertation, Cornell University 1988

[Pareles90] "The Midi Menace: Machine Perfection is Far From Perfect" *The New York Times* May 13, 1990

[Posner89] *Foundations of Cognitive Science* Michael I. Posner, ed. Cambridge : MIT Press 1989

[Rosenthal89] Rosenthal, D., "A Model of the Process of Listening to Simple Rhythms" *Music Perception* 6:3 Spring 1989 pp. 315-328

[Scarborough et al.89] Scarborough, D. Miller, B. and Jones, J., "Connectionist Models for Tonal Analysis" *Computer Music Journal* 13:3 1989 pp. 49-55

[Schank & Abelson77] Schank, R.C. & Abelson, R.P. *Scripts, Plans, Goals, and Understanding* Hillsdale NJ: Erlbaum 1977

[Schenker33] Schenker, H. *Five Graphic Music Analyses,* with an introduction by Felix Salzer New York: Dover Publications Inc. 1969 (first published 1933)

[Schenker35] Schenker, H. *Free Composition (Der freie Satz)* translated and edited by Ernst Oster New York: Longman 1979 (German edition first published 1935)

[Schwanauer88] Schwanauer, S., "Learning Machines & Tonal Composition" *Proceedings of the First Workshop on Artificial Intelligence and Music* AAAI-88 Minneapolis/St. Paul Minnesota August 24, 1988 pp. 34-45

[Serafine88] Serafine, M. L., *Music as Cognition: The Development of Thought in Sound* New York: Columbia University Press 1988

[Simon & Sumner68] Simon, H.A. & Sumner, R.K."Pattern in Music" in *Formal Representation of Human Judgement,* Benjamin Kleinmuntz, ed. 3rd Symposium on Cognition, Carnegie-Mellon 1967 New York : John Wiley & Sons. , Inc. 1968 pp. 219-250

[Sloboda85] Sloboda, J., *The Musical Mind: The Cognitive Psychology of Music* Oxford: Clarendon Press 1985

[Teitelbaum84] Teitelbaum, R., "The Digital Piano and the Patch Control Language System" *Proceedings of the ICMC 1984* pp. 213-216

[Vercoe84] Vercoe, B., "The Synthetic Performer in the Context of Live Performance" *Proceedings of the ICMC 1984* pp. 199-200

[Winograd&Flores86] Winograd, T., and Flores, F. *Understanding Computers and Cognition : A New Foundation for Design* Reading, MA : Addison-Wesley Publishing Company 1986

[Winston84] Winston, P., *Artificial Intelligence* (2nd edition) Reading, MA : Addison-Wesley Publishing Company 1984

[Zicarelli87] Zicarelli, D., "M and Jam Factory" *Computer Music Journal* 11:4 1987 pp. 13-29