1 Interactive Music Systems

Interactive computer music systems are those whose behavior changes in response to musical input. Such responsiveness allows these systems to participate in live performances, of both notated and improvised music. This text reviews a wide range of interactive systems, from the perspectives of several different fields. Each field, particularly music theory, artificial intelligence, and cognitive science, has developed techniques appropriate to various facets of interactive music systems. In relating a large number of existing programs to established bodies of work, as well as to each other, a framework for discussing both individual contributions and the field as a whole will be developed.

This book grew out of a doctoral thesis describing my own interactive system, called Cypher. For that reason, extensive consideration is given to the theoretical foundations and practical use of that program. Though the attention devoted to Cypher is out of proportion to that given the other programs reviewed here, the lattice of perspectives through which interactive systems will be viewed is more easily constructed with one example serving to focus the effort. Nonetheless, the book covers many other current applications in considerable detail, particularly when used in conjunction with the companion CD-ROM of audio and program examples. Further, basic concepts and characteristic issues will be illustrated using the Max programming environment, a graphic language for building interactive music systems (Puckette 1991).

1.1 Introduction

The use of computers has expanded musical thought in two far-reaching directions, the first of which concerns the composition of

timbre. Digital computers afford the composer or sound designer unprecedented levels of control over the evolution and combination of sonic events. The second expansion stems from the computer's ability to implement algorithmic methods for generating musical material. As in the case of timbral synthesis, the use of computers for algorithmic composition began with the earliest essays in the field (Koenig 1971). Recently, however, an important extension to this line of development has arisen from the realization of such processes in the context of live performance. By attaining the computation speeds needed to execute compositional algorithms in real time, current computer music systems are able to modify their behavior as a function of input from other performing musicians. Such changes of behavior in response to live input are the hallmark of interactive music systems. Interactivity qualitatively changes the nature of experimentation with compositional algorithms: the effect of different control variable values on the sounding output of the method can be perceived immediately, even as the variables are being manipulated (Chadabe 1989).

In this book, the musical motivations and possibilities accompanying interactive systems will occupy the forefront of the discussion. Second, I examine practical considerations of how to build, analyze, and extend these systems. Finally, I explore perspectives afforded by the viewpoints of artificial intelligence, cognitive science, and music theory in detail, both in terms of their actual contributions to the growth of the field to date, and of their potential impact on the continuing evolution of interactive music systems.

1.2 Machine Musicianship

In using a computer for composition or performance, the most fundamental question to be asked about any particular system is, What musical purpose does it serve? This book will cover several technical areas, some in considerable detail; however, the primary focus will be on the musical opportunities afforded by interaction and the ways in which these opportunities have been explored and elaborated by compositions and improvisations using them.

The responsiveness of interactive systems requires them to make some interpretation of their input. Therefore, the question of what the machine can hear is a central one. Most of the programs reviewed in this book make use of the abstraction provided by the MIDI standard (Loy 1985). An additional level of information can be gleaned from an analysis of the audio signal emitted by acoustic musical instruments. MIDI input and audio signals are low-level, weakly structured representations, which must be processed further by the program to advance any particular musical goals. How these low-level signals are interpreted and structured into higher-level representations is a research topic common to all interactive systems. The representations and processes available for constructing responses form the other broad area of inquiry.

Several of the systems we will review here interpret the input by emulating human musical understanding. Programming a computer to exhibit humanlike musical aptitude, however, is a goal with implications for a much broader range of applications. For almost any computer music program, in fact, some degree of musical understanding could improve the application's performance and utility. For example, in the automated editing of digital audio recordings, "simultaneous access to the low-level representations of the music in the signal and the higher-level constructs familiar to musicians would allow [automated editors] to perform operations and transformations whose realization by signal processing techniques alone would range from cumbersome to unimaginable" (Chafe, Mont-Reynaud, and Rush 1982, 537). With interactive systems, interpretation of the input is unavoidable. In non-real-time applications, such as sequencers or notation programs, an understanding by the program of concepts such as phrase, meter, direction, and so on could extend their function in the direction of a computer assistant, or interlocutor, able to suggest variations in tempo for the realization of a sequence or to locate points of significant change in a notated composition.

Capturing Musical Concepts

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. 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. Although research has begun to show us systematic ways in which human performers add expression to their rendering of a score (Palmer 1988), a general application of the fruits of this research will be impossible until 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 (Figure 1.1). 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.

In their interpretation of musical input, interactive systems implement some collection of concepts, often related to the structures musicians commonly assume. Each interactive system also includes methods for constructing responses, to be generated when particular input constructs are found. As methods of interpretation approach the successful representation of human musical concepts, and as response algorithms move toward an emulation of human performance practices, programs come increasingly close to making sense of and accomplishing an instruction such as "broaden the end of the phrase."

Before embarking on the formidable task of trying to achieve such behavior, one could ask why it is important for a computer program to approach human performance practices. In fact, throughout the early stages of electronic and computer music development, an expressed goal was often the elimination of human performers, with all their limitations and variability. "About 1920, when the slogan 'objective music' was in vogue, some famous composers (Stravinsky, for instance) wrote compositions specifically for pianola, and they took advantage of all the possibilities offered by the absence of restraints that are an outcome of the structure of the human hand. The intent, however, was not to achieve superior performance but to restrict to an absolute minimum the intervention of the performer's personality" (Bartok 1937, 291).

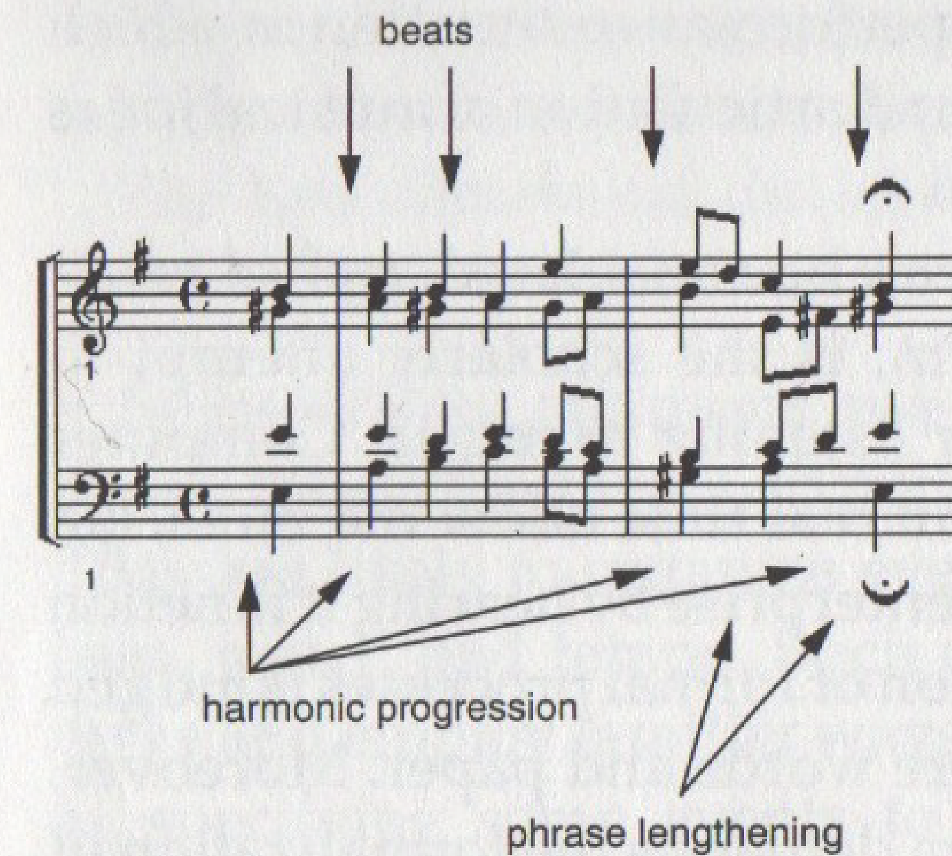

Figure 1.1

Computer music has, in fact, provided a perfect vehicle for eliminating the performer's personality. Compositions realized on tape can be painstakingly constructed by the composer, who in effect "performs" the work while entrusting it to a fixed realization, which is then played back without any further human intervention. Many of the most compelling and durable compositions in the field have been made in exactly this way. Eliminating performers entirely is hardly a desirable outcome, however, and one that few if any composers in the field would advocate. Their elimination is undesirable, beyond the purely social considerations, because human players understand what music is and how it works and can communicate that understanding to an audience, whereas computer performers as yet do not.

Works for performers and tape have been an expression of the desire to include human musicianship in computer music compositions. Coordination between the fixed realization of the tape and the variable, expressive performance of the human players, however, can become problematic. Such difficulties are more pronounced when improvisation becomes part of the discourse. And, as taped and performed realizations are juxtaposed, the disparity between levels of musicality evinced by the two often becomes untenable.

Composition by Refinement

Interactive music systems contribute to a process of composition by refinement. Because the program reacts immediately to changes in configuration and input, a user can develop compositional applica-

tions by continually refining initial ideas and sketches, up to the development of complete scripts for a performance situation in which the computer can follow the evolution and articulation of musical ideas and contribute to these as they unfold.

Further, many interactive systems can be considered *applied music theories*. Music theory, in its best form, is the scholarly attempt to describe the process of composing, or 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 it function in live musical contexts. The construction of formal processes is judged by the ear and sound, not through more words and paper. Moreover, implementation in a computer program demands the formalization of a theory to the point where a series of machine instructions can realize it. For suitable theories, the added rigor of realization by computer can clarify their formulation and make them available in a form from which they can be extended or used in other computational tasks. When a theory has been brought to the point of interactivity, it can be applied to the production and analysis of music in its native environment—that is, performed and experienced as live music has always been.

1.3 Classification of Interactive Systems

A primary objective of this book is to provide a framework within which interactive systems may be discussed and evaluated. Many of the programs developed to date have been realized in relative isolation from one another, with little scope for building on the work of earlier efforts. Now, several fundamental tools of the trade have become standardized and are no longer so subject to ad hoc solutions. Here I will propose a rough classification system for interactive music systems. The motivation for building such a set of classifications is not simply to attach labels to programs but to recognize similarities between them and to be able to identify the relations between new systems and their predecessors.

This classification system will be built on a combination of three dimensions, whose attributes help identify the musical motivations behind types of input interpretation, and methods of response. The dimensions will be described using some points along the continuum of possibilities for that dimension, points generally close to the extremes. The points used should not be considered distinct classes, however. Any particular system may show some combination of the attributes outlined here; however, these metrics do seem to be useful in identifying characteristics that can often distinguish and draw relations between interactive programs.

The first dimension distinguishes *score-driven* systems from those that are *performance-driven*.

- Score-driven programs use predetermined event collections, or stored music fragments, to match against music arriving at the input. They are likely to organize events using the traditional categories of beat, meter, and tempo. Such categories allow the composer to preserve and employ familiar ways of thinking about temporal flow, such as 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 anticipate the realization of any particular score. In other words, they do not have a stored representation of the music they expect to find at the input. Further, performance-driven programs tend not to employ traditional metric categories but often use more general parameters, involving perceptual measures such as density and regularity, to describe the temporal behavior of music coming in.

Another distinction groups response methods as being *transformative, generative*, or *sequenced*.

- Transformative methods take some 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. This material need not be stored, however—often such transformations are applied to live input as it arrives.
- For generative algorithms, on the other hand, what source material there is will be elementary or fragmentary—for example, stored scales or duration sets. Generative methods use sets of rules to produce complete musical output from the stored fundamental material, taking pitch structures from basic scalar patterns according to random distributions, for instance, or applying serial procedures to sets of allowed duration values.
- Sequenced techniques use prerecorded 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 *player* paradigms.

- Instrument paradigm systems are concerned with constructing an extended musical instrument: performance gestures from a human player are analyzed by the computer and guide an elaborated output exceeding normal instrumental response. Imagining such a system being played by a single performer, the musical result would be thought of as a solo.

- Systems following a player 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 player paradigm system played by a single human would produce an output more like a duet.

For the moment, a brief example will clarify how the dimensions are used. *Score followers* are a group of programs able to accompany a human instrumental soloist by matching her realization of a particular score against a stored representation of that score, simultaneously performing a stored accompanimental part. Such applications are a perfect example of score-driven systems. The response technique is sequenced, since everything the machine plays has been stored in advance. Finally, score followers can be regarded as player paradigm systems, because they realize a recognizably separate musical voice, assuming the traditional role of accompanist in the performance of instrumental sonatas.

2 Fundamentals

Two basic pillars of interactive systems are MIDI handling and scheduling. The Musical Instrument Digital Interface (MIDI) standard was developed by instrument and synthesizer manufacturers, and it allows controllers, computers, and synthesizers to pass data among themselves (Loy 1985). All of the systems reviewed here deal with the MIDI standard to some extent and therefore require software components to receive, package, and transmit properly formatted MIDI messages. The second imperative of music programs is to be able to perform tasks at specified points in time. Music is a temporal art, and any computer program dealing with music must have sophisticated facilities for representing time and for scheduling processes to occur at particular points in time.

The processing chain of interactive computer music systems can be conceptualized in three stages. The first is the *sensing* stage, when data is collected from controllers reading gestural information from the human performers onstage. Second is the *processing* stage, in which a computer reads and interprets information coming from the sensors and prepares data for the third, or *response* stage, when the computer and some collection of sound-producing devices share in realizing a musical output.

An important reason for dividing the chain into these three links is that there are usually machine boundaries between each successive stage. Commercial manufacturers dominate the sensing and response stages, through MIDI controllers and synthesizers. The sophistication of these devices, which implement several modes of operation and can often themselves be programmed, requires a discussion of sensing and response, rather than simply of a real-time computer program with some input and output. In full-blown interactive music systems, functionality is spread across three clusters of machines, and the designer often can choose to place certain methods within one or another cluster.

The processing stage has commercial entries as well, most notably MIDI sequencers. It is in processing, however, that individual conceptions of interactive music are most readily expressed, in any of a variety of programming languages with temporal and MIDI extensions. In this chapter, we will examine the three phases of interactive processing, looking at the hardware associated with each, the impact of machine boundaries separating each phase, and the nature of communications protocols developed to pass information between them.

2.1 Sensing

The recent fast growth in the development of interactive music systems is due in no small part to the introduction of the MIDI standard. We will begin our examination of the sensing stage with a consideration of MIDI, followed by a look at some of the standard's limitations and the way new sensing technologies have grown up to fill the gaps.

The MIDI Standard

The MIDI standard is a hardware specification and communications protocol that allows computers, controllers, and synthesis gear to pass information among themselves (Loy 1985). 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, that go on and off. The MIDI abstraction is eminently well suited to keyboard instruments, such as piano or mallet percussion, which can be represented as a series of switches. In fact, MIDI sensing on such instruments operates by treating each separate key as a switch. When the key is depressed, a Note On message is sent out, indicating which key was struck and with what velocity. When the key is released, a Note Off message (or, equivalently, Note On with a velocity of zero) is transmitted with the key number.

The note concept is the fundamental MIDI paradigm. All MIDI instruments implement it. Following its genesis from commercial keyboard controllers, MIDI represents continuously varying control functions as well. One of these is known as *pitchbend*, usually implemented with a wheel built into the instrument. Each time the wheel is moved, a new pitchbend value is transmitted to the processing stage. Such continuous control messages are transmitted with a controller number and a value. Many instruments allow the user to assign controller numbers to physical devices, remapping the pitchbend wheel, for example, to another control channel. The generation of continuous controls is therefore, in most cases, easily reconfigurable. Moreover, additional layers of remapping and interpretation can be implemented in the processing and response stages.

Although the introduction of the MIDI standard has had the salutary effect of greatly expanding research into and performance with interactive music systems, use of the standard imposes limitations of several kinds (Moore 1988). First, because MIDI communicates on the control level—performance gestural events rather than any representation of the audio signal—the standard cannot be used to describe or control much of the timbral aspect of a musical performance. Control over synthesis and the evolution over time of any particular sound is coded into each synthesizer and may be affected through MIDI only by using an ad hoc collection of triggers, continuous controls, and system exclusive commands, private to each machine and inimical to the very idea of a standard.

That said, several synthesis algorithms have been built into commercial synthesizers that are able to achieve a quite broad and subtly varying range of sounds. The most successful of these offer a small number of fairly powerful control variables, which fit the control situation of a MIDI environment well. In a typical transaction, a Note On message emitted from a computer or controller will trigger a complex reaction from the synthesizer, realizing the attack portion of the sound. Continuous controls can affect variables of the synthesis algorithm during the "steady state" portion of the sound, modifying such things as filter cutoff frequencies or the amplitude of a modulating oscillator. A Note Off message then initiates the sound's decay, again stored as a complex, though relatively constant event in the synthesizer.

A primary cause of this transaction paradigm is the fact that the MIDI standard enforces a transmission bandwidth of 31,250 bits per second. Each eight-bit MIDI byte is surrounded with start and stop bits, making their effective size equal to ten bits. Therefore, a standard MIDI Note On message, which requires three bytes (30 bits) of information, takes approximately 1 millisecond to transmit. As Gareth Loy points out in Loy 1985, the performance through MIDI of a ten-note chord will introduce a delay of 10 milliseconds between the first note of the chord and the last. Though 10 milliseconds is not enough to affect the percept of that event as a single chord, it can have an effect on the timbral

quality of the sound. Further, when we consider the impact of a 1-millisecond transmission time on the performance of a ten-note chord sent out simultaneously over ten channels, the 100-millisecond delay between the first note and the last will certainly be heard.

When we consider MIDI as a communications channel not only for note messages but for the state of continuous controllers, the bandwidth problems are even more serious. Though the standard does not provide a good way to control the internal evolution of a sound, such facilities as there are cannot be fed quickly enough at 31,250 bits per second to afford a performer close control over the sounding result (Moore 1988). The result is to accentuate the machine boundary between the computer and its outboard synthesis gear: a user must program the synthesizer with instructions concerning how to evolve a sound in response to triggers sent out from the computer. The limitations of MIDI bandwidth are thus somewhat attenuated, moving some of the musical control over to the sound gear, but at the expense of an integrated and flexible development environment for all aspects of computer performance.

Despite these concerns, the positive influence of the MIDI standard has far outweighed its limitations. Most of the work described in this book could not have been achieved without it. As interactive systems grow in power and application, we can only hope that the standard will grow with them. In any event, the migration of sensing and response capabilities back to the host computer has the potential to obviate the need for such a communications protocol entirely.

Custom Controllers

Information can arrive from sensors in forms other than MIDI. Another important input type is *samples*, the digital representation of a time-varying audio signal. Usually, the sound of some musical instrument is picked up by a microphone, sent through an analog-to-digital converter (ADC), and then on to the computer. Compact-disc-quality sampling rates for digital audio produce (at least) 44,100 16-bit samples per second, so dealing with a raw sample stream demands very high-powered processing. For that reason, interactive systems designed to handle audio have dedicated hardware devices able to process sounds at the requisite speeds. We will return to a discussion of audio input and output in the section on response.

MIDI controllers can be thought of as gestural transducers, providing a representation of human musical performance. Keyboard performances are represented fairly well. Much of the important gestural information from performance on other instruments (strings, winds, voice), however, is not fully captured by MIDI sensors. For that reason, a significant research effort has grown up around the attempt to build controllers better able to capture the range of expression afforded by traditional musical instruments.

Implementing the violoncello interface used for Tod Machover's composition *Begin Again Again . . .*, a team including Neil Gerschenfeld, Joseph Chung, and Andy Hong developed sensors to track five aspects of the cellist's physical performance: (1) bow pressure, (2) bow position (transverse to the strings), (3) bow placement (distance from the bridge), (4) bow wrist orientation, and (5) finger position on the strings (Machover et al. 1991). To sense positions (2) and (3), a drive antenna was mounted on the cello bridge and a receiving antenna on the bow. The capacitance between the two yielded both position of the bow across the strings and position of the bow relative to the bridge. Pressure on the bow was measured from the player's finger, rather than the bow hairs. Again a capacitance measurement was used, by putting a foam capacitor around the part of the bow where finger pressure would be applied. Finger position on the strings was found from the resistance between the metal strings and strips of conductive thermoplastic sheet mounted on the fingerboard beneath them. Finally, wrist angle was read from a sensor mounted on the wrist that measures joint angles from the movement of magnets corresponding to the wrist and the back of the hand. These five traits of the physical gestures were continually tracked by computers during the performance of the piece, and used to change the timbral presentation of computer music associated with each section of the composition (see also section 3.4).

Space-Control Performance

Using gestural control to affect the output of electronic instruments is a principle that was already firmly established in 1919 by the Russian scientist Lev Termen with his "Aetherphone," later called the "Theremin" after the Gallicized version of his name (Glinsky 1992). The instrument produces monophonic music with a quasi–sine wave timbre and is

played by moving two hands in the vicinity of antennae controlling pitch and amplitude. The tones come from two heterodyning oscillators, one of fixed frequency, and the other of variable frequency. Both oscillate at frequencies well above the range of human hearing; however, when the variable oscillator's frequency is changed, a difference tone is created between it and the reference oscillator, and this difference tone does fall in the audible range. Moving the right hand toward and away from one antenna changes the variable oscillator frequency, thereby producing eerie portamento effects as the audible beat frequency goes up and down. The amplitude is similarly controlled by the movements of the other hand. Because of this continuous control over pitch and loudness, the instrument is capable of quite expressive performances, though mastery of it requires years of practice. The Theremin was a sensation, and it was played by its inventor and other virtuoso performers to packed houses throughout Europe and the United States. Similar devices existed around the same time but never caught the public imagination the way the Theremin did. An important reason for this was what Termen called "space-control performance": the fact that the instrument was played without anyone actually touching it. Competitors outfitted with more conventional keyboard or other interfaces never aroused the same sense of wonder as the space-control Theremin.

A fascination with the seeming magic of music performed by movement of the hands alone has carried through to the use of hand-based controllers in several recent interactive systems. Stichting STEIM in Amsterdam has devoted a considerable research effort to the development of new gestural controllers, resulting in such devices as The Hands (a pair of proximity-sensitive hand-mounted sensors) and The Web (a weblike device in which manipulations of one part affect other regions) (Krefeld 1990). Similarly, a number of hand controllers have been marketed for various purposes, including the Exos Dextrous Hand Master, the VPL Data Glove, and the Mattel Powerglove, which have subsequently been adapted for experimentation in interactive music. Two composers who have used this technology are Michel Waisvisz of STEIM, who performs the composition *The Hands* with the controller of the same name, and Tod Machover, who adapted the Dextrous Hand Master to control timbral variation through MIDI mixers in his composition *Bug-Mudra*.

The Buchla Lightning Controller is another variation on the same idea: the controller responds to motions made in space with various transmitting devices (Rich 1991). One of these transmitters is a wand, which sends infrared information to a control box. The control box is thereby able to track the motion of the wand within a performance field, which is about as wide as the distance from the transmitter to the control box and whose height is about 60 percent of the width. This performance field is split up into eight cells, and various combinations of the eight cells make up different zones. The controller responds to "strikes" (quick changes of direction made with the transmitter) within a zone, entry or exit from any zone, and combinations of switches built into the transmitters and foot pedals. The control box can be programmed to send out MIDI note, control, or program change commands, as well as MIDI clock messages. Further, one of the Lightning presets allows the controller to talk directly to Max patches. Lightning is thus a general control device, which enables users to make physical gestures in the air and define the functionality of those gestures through MIDI-based programming.

The Radio Drum is a three-dimensional percussion controller, sensitive to the placement of beats on the face of the drum and the position of the drumstick through the air as it approaches the drum face (Mathews 1989). The Radio Drum was designed by Max Mathews and Robert Boie, and Mathews's main conception of the device is to use it to control tempo. In one performance, a singer beats the Radio Drum to cue each successive attack in the performance of an accompanimental part. Of course, the three-dimensional information coming from the drum can be used to trigger much more intricate interactions between the drummer and the computer, an area which has been extensively explored by such composers as Andrew Schloss and Richard Boulanger.

Pitch Detectors

Solutions geared to individual instruments have proliferated because of significant differences among the instruments themselves: critical controls for one family of instruments do not even exist on others. Another obstacle is the fact that reliable real-time pitch tracking from an arbitrary audio signal has not been developed. Commercial devices for the job (pitch-to-MIDI converters) exist, but they have widely variable results for different instruments, even for different performers on the same instrument. General solutions have not been forthcoming because the problem is a difficult one: techniques such as the Fast Fourier Transform (FFT) are often not fast enough or simply fail to find the correct pitch. Consider the case of trying to identify a pitch at 60 hertz: generally, two cycles of the waveform will be required for

analysis. At that frequency, a delay of over 30 milliseconds is required in the best case for identification. When we consider, in addition, that the attack portion of an instrumental waveform will be the least regular part of it, the difficulty of quickly and accurately finding the pitch through standard Fourier analysis is readily seen.

One approach to the pitch-tracking problem has been to solve it for other musical instruments as it was solved for keyboards. The keyboard, and mallet percussion, case is simple: each key is treated as a switch, and the pitch played can be read off the switch depressed. Extending this concept to other instruments means fitting them with mechanical sensors to read fingerings or hand position, reducing the problem of pitch detection to one of indexing known fingerings to the pitches they produce. In cases where one fingering produces more than one pitch (overblown wind instruments, for example), minimal additional signal processing can disambiguate between the remaining possibilities.

An early instance of fingering-based pitch detection was built into the IRCAM flute controller. Optical sensors tracked the manipulations made by the player's fingers on the instrument, and the found configurations were used as indices into a table of known pitches for each fingering. Since one flute fingering can potentially produce more than one pitch, additional signal processing on the 4X machine analyzed the audio signal to decide between the remaining possibilities (Baisnee et al. 1986). Other wind instrument devices have been built following similar physical-mapping principles: the Yamaha WX-7 wind controller follows key positions, and in fact extends the traditional range of single-reed instruments with additional octave keys, allowing the player to cover up to seven octaves.

MIDI adaptations for other instruments abound as well. MIDI guitars, for example, have been built by several manufacturers. Entire families of string instruments have been commercialized by Zeta and the RAAD group. Hardly an instrument exists that has not had some work done to allow it to transmit MIDI messages. The underlying message of this trend is clear: the motivation to participate in the expanded possibilities afforded by computer-based instruments is a compelling one for performers and builders alike. The general operating principles of existing orchestral instruments are being maintained, to capitalize on the years of training professional players have had. Because traditional instruments are markedly different, new control adaptations must be made on a highly individual basis. General analysis systems based on the properties of audio signals have not provided a solution, given the inability of techniques such as Fourier analysis to provide accurate results within the demands of real-time performance. Given these constraints, new instrumental controllers combine some interpretation of physical gestures with audio signal processing to provide a wider range of gestural information from instrumental performance. At the same time, controllers divorced from any relation to the orchestral instruments, but which give wider scope to the tracking of expressive gestures, are being developed in growing numbers.

2.2 Processing

The information collected during the sensing stage is passed on to a computer, which begins the processing stage. Communication between the two stages assumes a protocol understood by both sides, and the most widely used protocol is the MIDI standard. Other signals passing between them could include digital audio signals or custom control information. In either case, some form of the procedure for handling MIDI streams would apply.

Early interactive systems always needed to implement a MIDI driver, which was capable of buffering a stream of serial information from a hardware port and then packaging it as a series of MIDI commands. Such a facility still must be included in every program, but for most hardware platforms the problems of MIDI transmission have been solved by a standardized driver and associated software. A good example is the Midi Manager™, written for Apple Macintosh computers. The Midi Manager makes it possible for several MIDI applications to run simultaneously on a single computer. An associated desk accessory called PatchBay allows a user to route MIDI streams between applications and the Apple Midi Driver, and to specify timing relations between all of them (Wyatt 1991).

Beyond the bookkeeping details of receiving and packaging valid MIDI commands, MIDI drivers invariably introduce *time stamps*, a critical extension of the standard. Time stamps are an indication of the time at which some MIDI packet arrived at the computer. In the case of the MIDI Manager, time stamps are notated in one of a number of possible formats, with a resolution of one millisecond. With the addition of time stamps, the timing information critical to music production becomes available. Interpretation processes can use this information

to analyze the rhythmic presentation of incoming MIDI streams. Timing information is also needed as the computer prepares data for output through the response stage, and it is here that the function of a real-time scheduler comes into play.

Real-Time Schedulers

Scheduling is the process whereby a specific action of the computer is made to happen at some point in future time. Typically, a programmer can invoke scheduling facilities to delay the execution of a procedure for some number of milliseconds. Arguments to the routine to be executed are saved along with the name of the routine itself. When the scheduler notices that the specified time point has arrived, the scheduled process is called with the saved arguments.

Much of the work on interactive music systems developed at the MIT Media Laboratory, for example, relies on a scheduler adapted from work described in (Boynton 1987). 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 records the number of centiseconds that have passed since the MIDI driver was opened. Accordingly, incoming 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 are associated with 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 that 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*: 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 that 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.

In figure 2.1, 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 nonzero *period* argument is included, the task will reschedule itself *period* centiseconds after each execution. Periodicity is a widespread attribute of music production, and the facility of the *period* argument elegantly handles the necessity. Tasks that are periodically rescheduling themselves can be halted by an explicit kill command at any time. In figure 2.1, 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.

A similar facility is provided by the CMU MIDI Toolkit, adapted from Doug Collinge's language *Moxie* (Dannenberg 1989): the cause()

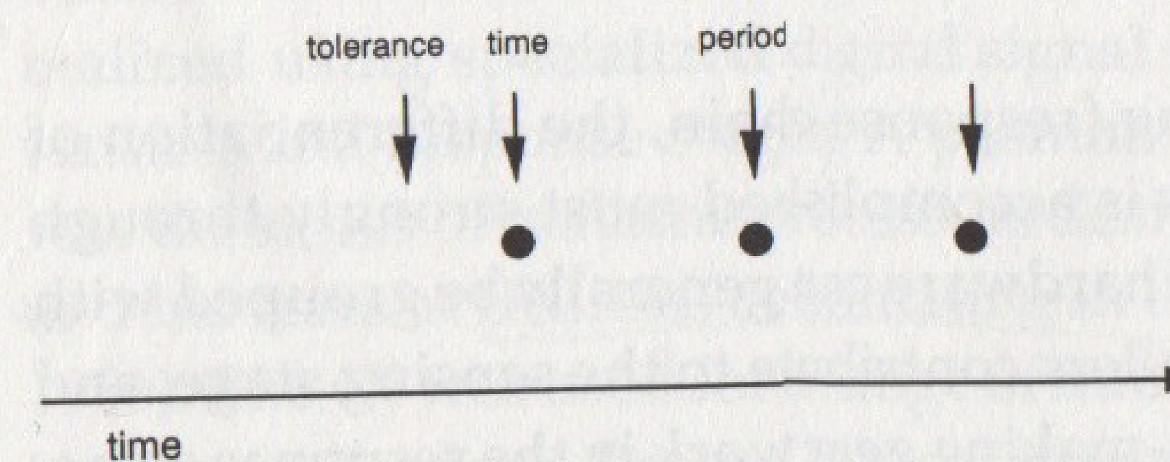

Figure 2.1

routine will invoke a procedure, with the listed arguments, some number of centiseconds in the future. For instance, the call

```
cause(100, midi_note, 1, 60, 0);
```

will induce the scheduler to call the routine midi_note 100 centiseconds after the execution of cause(). The arguments 1, 60, and 0 are specific to midi_note (and correspond to channel, note number, and velocity). Any list of arguments can be presented after the name of the routine to be invoked, as in the Boynton syntax shown above. In fact, one of the most noticeable differences between the two schedulers is that Boynton's expects absolute time points as a reference, whereas Dannenberg's cause() routine requires the execution time of the scheduled routine to be specified relative to the invocation of *cause* itself.

For both versions of this idea (and there are several others), it is important to note that the model assumes that none of the scheduled routines will require a long processing time. Once a procedure is called by the scheduler, it runs until the end, and then relinquishes control to the scheduler again. These routines are not interrupted, except by a strictly limited number of input handlers. If some procedure requires extensive processing, it must reschedule itself at regular intervals to allow the scheduler enough CPU time to keep up with the execution of any other tasks in the queue.

An extensive literature has grown up around the subject of real-time scheduling, particularly as it affects music programming, and several languages have been implemented that address the problem in various ways. The reader is referred to Roger Dannenberg's excellent survey (Dannenberg 1989). Besides the scheduler included in the Midi Manager, extensive possibilities for scheduling events have been built into Max, the graphic programming language for interactive systems that we will examine in some detail.

Building Functionality

Along the sensing/processing/response chain, the differentiation of functionality among systems is accomplished most strongly through processing. Certain classes of hardware can generally be grouped with each link in the chain. Controllers contribute to the sensing stage, and synthesizers and other sound-making gear work in the response stage. The hardware of the processing stage is a digital computer. The programs running in this computer are the subject of this book, and different approaches to the processing stage will be seen repeatedly in the following pages.

We have already reviewed some of the problems surrounding sensing and two of the indispensable components of processing: MIDI handling and scheduling. The rest of the processing component is what distinguishes Michel Waisvisz's *The Hands* from Morton Subotnick's *A Desert Flowers* from Jean-Claude Risset's *Duet*. In describing the sequence of events in an interactive music system, this subsection provides only a placeholder for a discussion of the cornucopia of possible processing approaches. The rest of this book fills the placeholder by describing in detail the processing link in the chain.

2.3 Response

The response stage resembles the sensing stage most strongly in the protocol used to pass information. Again here, the computer and the devices used to actually perform the responses communicate most often through the MIDI standard. The exact nature of the directions sent out by the computer will depend on the synthesis devices in use and the kinds of effects those devices are used to realize. Presently, commercial MIDI gear tends to fall into two large groups: synthesis and sampling. Synthesis modules use an algorithm—for example, frequency modulation—to produce sounds. Sampling gear has stored waveforms, often recordings of traditional acoustic instruments, which are played back at specific pitches in response to MIDI messages. Response commands sent to the devices, then, would include Note On and Off messages and whatever controller values are desired for manipulation of the device's sound production technique.

Real-Time Digital Signal Processing

Before the arrival of MIDI, interactive computer music was often realized using specialized digital signal processing hardware for the sensing and response stages. A prominent example of this approach was the series of realtime processors designed by Giuseppe di Giugno and his team at IRCAM, culminating in the 4X machine (Baisnee et al. 1986). The 4X was used for a range of interactive compositions, including Pierre Boulez's *Repons*, several pieces by Philippe Manoury, such as *Jupiter* and *Pluton*, and my own *Hall of Mirrors*. Sensing involved the intake of audio samples from microphones and reading commands

typed at an alphanumeric keyboard. Responses were generated by real-time signal processing of the audio taken in from the microphones, producing a live transformation of sounds already being performed by acoustic instruments. Later, MIDI handling was added to the capabilities of the 4X real-time system, to take advantage of the MIDI controllers which were by then becoming available in large numbers (Favreau et al. 1986).

As MIDI synthesis equipment became more sophisticated and duplicated many of the techniques developed in the leading computer music institutions, much of the work of response generation fell to such devices. Another important reason for the dominance of commercial gear through the late 1980s and early 1990s was the considerable expense involved in acquiring and maintaining a machine like the 4X, a cost prohibitive for most studios and certainly for individuals.

With the introduction of digital signal processing chips such as the Motorola 56000, however, the pendulum began to swing back toward real-time signal processing as a viable choice for generating sound. The efficiencies of mass production, coupled with the installation of these chips on a variety of add-on boards designed for personal systems such as the IBM PC, Apple Macintosh, and NeXT machine, made DSP hardware both inexpensive and relatively widespread. The viability of digital signal processing power in personal workstations is changing the face of response synthesis techniques. Direct-to-disk sampling applications, and specialized hardware/software packages such as the Digidesign SampleCell, now make possible the extensive use in performance of recorded sound, or transformations of live sound, without relying on external MIDI samplers.

IRCAM Signal Processing Workstation

The accelerated use of digital signal processing in live performance is again changing the nature of response capabilities in interactive music systems. The standard for the integration of control- and audio-level programming has moved forward with the commercialization of the IRCAM Signal Processing Workstation (ISPW). The ISPW consists of a NeXT computer equipped with a special accelerator board, on which reside two Intel i860 processors (Lindemann et al. 1991). The i860s are very fast, general-purpose processing devices. Before the ISPW, real-time digital signal processing tasks were accomplished with specialized processors, built particularly for DSP, such as the Motorola 56000. The advantage of using a general-purpose processor such as the i860

is that the machine boundary between signal- and control-level computations is erased.

To take advantage of this flexibility, a new version of Max was written to include signal objects. These objects can be used to build signal-processing programs, just as MIDI Max objects are configured to implement control programs. When the two classes are combined, the conceptualization and implementation of interactive systems using real-time signal processing is considerably eased. First, the response phase of the system is entirely programmable: not the choices of a manufacturer but the demands of a composition can decide the synthesis algorithms used. Second, a single programming environment can be used for both the processing and response phases. Third, since processing and response are realized with the same machine through the same programming environment, the need for communications protocols such as MIDI, with all their bandwidth and conceptual limitations, falls away (Puckette 1991).

The realization on the ISPW of powerful signal analysis techniques can eliminate much of the need for external sensing as well. The workstation is fast enough to perform an FFT and inverse FFT in real time, simultaneously with an extensive network of other signal and control processing. Already pitch- and envelope-tracking objects have been used for compositional sketches. If continuous sensing of pitch, amplitude, and timbral information can be achieved from the audio signal alone, the entire sensing/processing/response chain could be reduced to a single machine, with all the attendant gains in flexibility and implementation power that entails.

2.4 Commercial Interactive Systems

Commercially available interactive systems began to appear in the mid-1980s. Such programs illustrate the processing chain outlined in the previous section and several hallmarks of interaction. Rather than survey the full range of applications, we will briefly consider *M* and *Jam Factory*, two ground-breaking efforts in the field, before moving on to Max, a newer graphic programming environment that allows users to design their own interactive systems.

M and Jam Factory

In December of 1986, Intelligent Music released M and Jam Factory. Intelligent Music is a company founded by composer Joel Chadabe for

developing and distributing interactive composing software. Chadabe and three others designed M; one of those collaborators, David Zicarelli, also designed Jam Factory (Zicarelli 1987). Among the breakthroughs implemented by these programs are graphic control panels, which allow access to the values of global variables affecting their musical output. Manipulating the graphic controls has an immediately audible effect. The sensing performed by M and Jam Factory centers around reading manipulations of the control panel and interpreting an incoming stream of MIDI events. Responses are sent out as MIDI.

Each program implements different, though related, compositional algorithms. "The basic idea of M is that a pattern (a pattern in M is a collection of notes and an input method) is manipulated by the various parameters of an algorithm" (Zicarelli 1987, 19). Four patterns are active simultaneously, and variables for each of them can be changed independently. The algorithm allows control over such parameters as *orchestration*, which assigns patterns to MIDI channels; *sound choice*, which selects program changes for the channels of a pattern; *note density*, the percentage of time that notes from a pattern are played; and *transposition*, which offsets pitch material from its original placement. Duration, articulation, and accent are governed by *cyclic distributions*, collections of data used to reset the values of these parameters with each clock tick. Further, the mouse can be used to "conduct" through various settings of these variables.

Jam Factory implements four *players*, whose material is generated using the transition tables characteristic of Markov chains (see section 6.2). Several tables of different orders are maintained, and "an essential part of the Jam Factory algorithm is the probabilistic decision made on every note as to what transition table to use" (Zicarelli 1987, 24). Separate transition tables for pitches and durations are employed, and each player has independent tables for both parameters. The probabilities of different-order transition tables being used affects the degree of variation and relatively straightforward playback of the stored material. For instance, first-order tables depend only on the previous pitch (in the case of melodic generation) to determine the following one. Second-order tables look at the previous two pitches, and so on. "For many applications, 70-80 percent Order 2 with the rest divided between Orders 1 and 3 will blend 'mistakes' with recognizable phrases from the source material in a satisfying manner" (Zicarelli 1987, 25).

Timing is expressed on two levels. First, a master tempo defines the rate of clock ticks. Then a time base, independent for each voice, sets the number of ticks that pass between successive events. This scheme closely follows the MIDI conception of time, which is organized around beats in a tempo. Beats always represent an identical number of ticks: quarter notes, for example, can span 24, 48, or 96 clock ticks. Though the number of ticks is variable for different applications or pieces of hardware, once a resolution is chosen, it remains constant for that notated duration. The speed of quarter-note realization, then, is changed by varying the overall tempo, which defines the duration of one clock tick. The advantage of this scheme is that the relation of events to an underlying meter is kept constant through tempo changes. The disadvantage is that it enforces a conceptualization of musical time in beats and tempi, even in music for which these are not appropriate categories.

M and Jam Factory are clear examples of performance-driven interactive systems. There is no stored score against which input is matched. The performance driving the program is basically a series of gestures with the mouse; an *input control system* allows the same functionality to be transferred to a MIDI keyboard. The response method is generative: stored lists of material are varied through the manipulation of a number of performance parameters. The programs follow a player paradigm in that they realize a distinct musical voice from the human performance. In fact, M and Jam Factory are unusual with respect to later programs in that the human performance itself is not particularly musical: rather, the performer's actions are almost entirely directed to manipulating program variables.

Max

In 1990, Opcode Systems released the commercial version of Max, a graphic programming environment for interactive music systems. Max was first developed at IRCAM by Miller Puckette and prepared for commercial release by David Zicarelli. Max is an object-oriented programming language, in which programs are realized by manipulating graphic objects on a computer screen and making connections between them. The collection of objects provided, the intuitively clear method of programming, and the excellent documentation provided by Opcode make Max a viable development environment for musicians with no prior technical training.

Throughout this book, examples of music programming techniques will be illustrated with Max. From the CD-ROM supplement, working

beat or tempo employed here, only time offsets. Second, it is essentially generative. Although external data feed the algorithm, the basic operation uses stored elemental material to generate the output, as opposed to transformations performed on the input as a musical whole. The system follows the player paradigm: the machine's music is recognizable as a voice separate from the human player.

The compositional method, in this form, is not particularly interesting. The output shows some readily perceptible relation to a human performance but does nothing more than repeat it with a constant, rudimentary style of variation. There are no control variables to allow changes in the variation technique. Many isorhythmic motets, however, are very beautiful pieces of music. This is because the formalism was again replete with "trapdoors": the isorhythmic method was used as a structural device, serving to unify and organize a much more complex and varied musical whole.

Often such formalisms have served a similar role—as a basic structure, a stimulant for the composer's imagination, or even an "in joke" between the composer and generations of analysts to come. As computers have been enlisted to realize such processes, however, more and more responsibility is given over to the process. Certainly, many programs are used collaboratively by the composer, who is free to override, interpret, or elaborate the output of the compositional formalism in any way, just as the users of isorhythmic techniques did. But as experience and a realization of the power of such processes grow, a great many programs have been implemented, with a sophisticated and highly developed sense of musicianship, for the automated, and unassisted, composition of music.

3 Live Computer Music

This chapter reviews a broad spectrum of existing interactive systems, elaborating and refining a framework for discussion of these programs in the process. Many of the fundamental issues surrounding interactive systems will be sharpened as they are considered in connection with working examples. Further, a range of compositional concerns and motivations come to light through a careful consideration of the music actually made.

3.1 Cypher

Cypher is an interactive computer music system that I wrote for composition and performance. 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 them 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 that 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 that may range from carefully notated performance instructions to more indeterminate descriptions of desired musical behavior. Particularly in the case of pieces that

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 that 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 until performance, to the point of using the program for 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.

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. Although 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 noticed in 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 that affect the generation of material through groups of several events.

The preceding description provides a rough sketch of the way Cypher works. Much more detail will be provided in later chapters: from the theoretical underpinnings in chapter 4 through an exposition of the analytical engine in chapter 5, to a review of composition methods in chapter 6. Already, however, we can relate the program to the descriptive framework for interactive systems.

Considered along the first dimension, Cypher is clearly a performance-driven system. The program does not rely on stored representations of musical scores to guide its interaction with human performers. A cue sheet of score orientation points is used in performances of notated compositions, but this level of interaction is distinct from the more rigorous following techniques of score-driven programs. Further, the facilities for playing back stored scores in response are quite attenuated in Cypher. Although stored fragments can be played back in a limited way, I have never used any sequences in compositions using Cypher.

The indistinct nature of the classification metrics I propose can be clearly seen by the position of Cypher with respect to the second dimension: the program uses all three techniques, though with varying degrees of emphasis, to produce responses. Of these three classes, Cypher performs transformations most often, algorithmic generation somewhat, and sequencing hardly at all.

Finally, Cypher is a player paradigm system. Performing with the program is like playing a duet or improvising with another performer. The program has a distinctive style and a voice quite recognizably different from the music presented at its input. The player/instrument paradigm dimension is again a continuum, not a set of two mutually exclusive extremes. We can place Cypher at the player paradigm end of the scale, however, because a musician performing with it will experience the output of the program as a complement, rather than as an extension or elaboration, of her playing.

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.

3.2 Score Following and Score Orientation

Now that we have seen how Cypher is used, we will review music that has been made with it. Before doing so, let us set the stage for that review, and the examination of other interactive systems following it, by looking at ways to coordinate machine and human performances. Two divergent means to the same end are considered: score following allows a tight synchronization between a solo part and a computer accompaniment; score orientation refers to a range of applications that coordinate human and computer players by lining them up at certain cue points in a composition.

Score Followers

The first dimension in our interactive system classification scheme distinguishes between score-driven and performance-driven programs. The paradigmatic case of score-driven systems is the class of applications known as *score followers*. These are programs able to track the performance of a human player and extract the tempo of that performance on a moment-to-moment basis. The best theory of the current tempo is then used to drive a realization of an accompaniment (Vercoe 1984; Dannenberg 1989).

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 the accompaniment accordingly.

The classic example of score following uses a computer to accompany a human soloist in the performance of a traditional instrumental sonata. As the human soloist plays, the computer matches that rendition against its representation of the solo part, derives a tempo, and plays the appropriate accompaniment. The human can speed up or slow down, scramble the order of notes, play wrong notes, or even skip notes entirely without having the computer accompanist get lost or play at the wrong speed. Such applications are unique in that they anticipate what a human player will do: the derivation of tempo is in fact a prediction of the future spacing of events in a human performance based on an analysis of the immediate past.

Score followers are score-driven, sequenced, player-paradigm systems. The pattern matching must be reliably fault tolerant in the face of wrong notes, skewed tempi, or otherwise "imperfect" performances rendered by the human player. Barry Vercoe's program 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 rendering of the piece (Vercoe 1984).

The application described in Dannenberg and Mont-Reynaud 1987 is able to perform real-time beat tracking on music that is not represented internally by the program. The beat-tracking part develops a theory of a probable eighth-note pulse, and modifies the tempo of this pulse as additional timing information arrives. At the same time, the harmonic section of the program navigates its way through a chord progression by matching incoming pitches from a monophonic solo against sets of the most probable pitches for each successive chord. Here, the principles of score following are expanded to implement a similar functionality—performing an accompanimental part against an improvisation. Some things about the improvisation are known in advance—for example, the chord progression—however, the precise sequence of notes to be played, and their timing relations, are not known. Here we can see the essence of score following, comparing a performance against a stored representation of some expected features of that performance, carried out on a level other than note-to-note matching. Such a program moves over toward the performance-driven end of the scale and demonstrates one potential application area for these techniques that goes beyond the classic case.

The musical implications of score following in its simplest form are modest: they are essentially tempo-sensitive music-minus-one machines, where the computer is able to follow the human performer

rather than the other way around. In most compositional settings, however, score following is used to coordinate a number of other techniques rather than to derive the tempo of a fixed accompaniment. In many of the compositions recently produced at IRCAM, for example, the computer part realizes a sophisticated network of real-time signal processing and algorithmic composition, guided throughout the performance by a basic score-following synchronization system. In his paper "Amplifying Musical Nuance," Miller Puckette describes this technique of controlling hidden parameters as a human player advances through the performance of a score (Puckette 1990). The hidden parameters are adjusted, and sets of global actions executed, as each successive note of the stored score is matched.

> The notes, and more especially the global actions, need not merely select synthesizer parameters. In practice, much use is also made in this context of live "processes" which may generate a stream of notes or controls over time; global actions can be used to start, stop, and/or parameterize them. Notes may themselves be used to instantiate processes, as an alternative to sending them straight to the synthesizer. If a "process" involves time delays (such as in the timing of execution of a sequence), this timing may be predetermined or may be made dependent on a measure of tempo derived from the live input. (Puckette 1990, 3)

The following section describes some of the basic tools in Max used to accomplish the kinds of score synchronization previously described.

Follow, Explode, and Qlist

In Max, several objects support score following, including *follow*, *explode*, and *qlist*. The *follow* object works like a sequencer, with the added functionality that a follow message will cause the object to match incoming Note On messages against the stored sequence. Matches between the stored and incoming pitches will cause *follow* to output the index number of the matched pitch. *Explode* is similar to *follow*, with a graphic editor for manipulating events. *Qlist* is a generalized method for saving and sending out lists of values at controlled times.

In the patch in figure 3.11, the outlets of *notein* are connected to *stripnote*, yielding only the pitch numbers of Note On messages. These are sent to a *makenote/noteout* pair, allowing audition of the performance (albeit a rather clumsy one, since all the velocities and durations are made equal by the arguments to *makenote*). Further, the incoming pitch numbers are sent to a *follow* object, to which are connected a

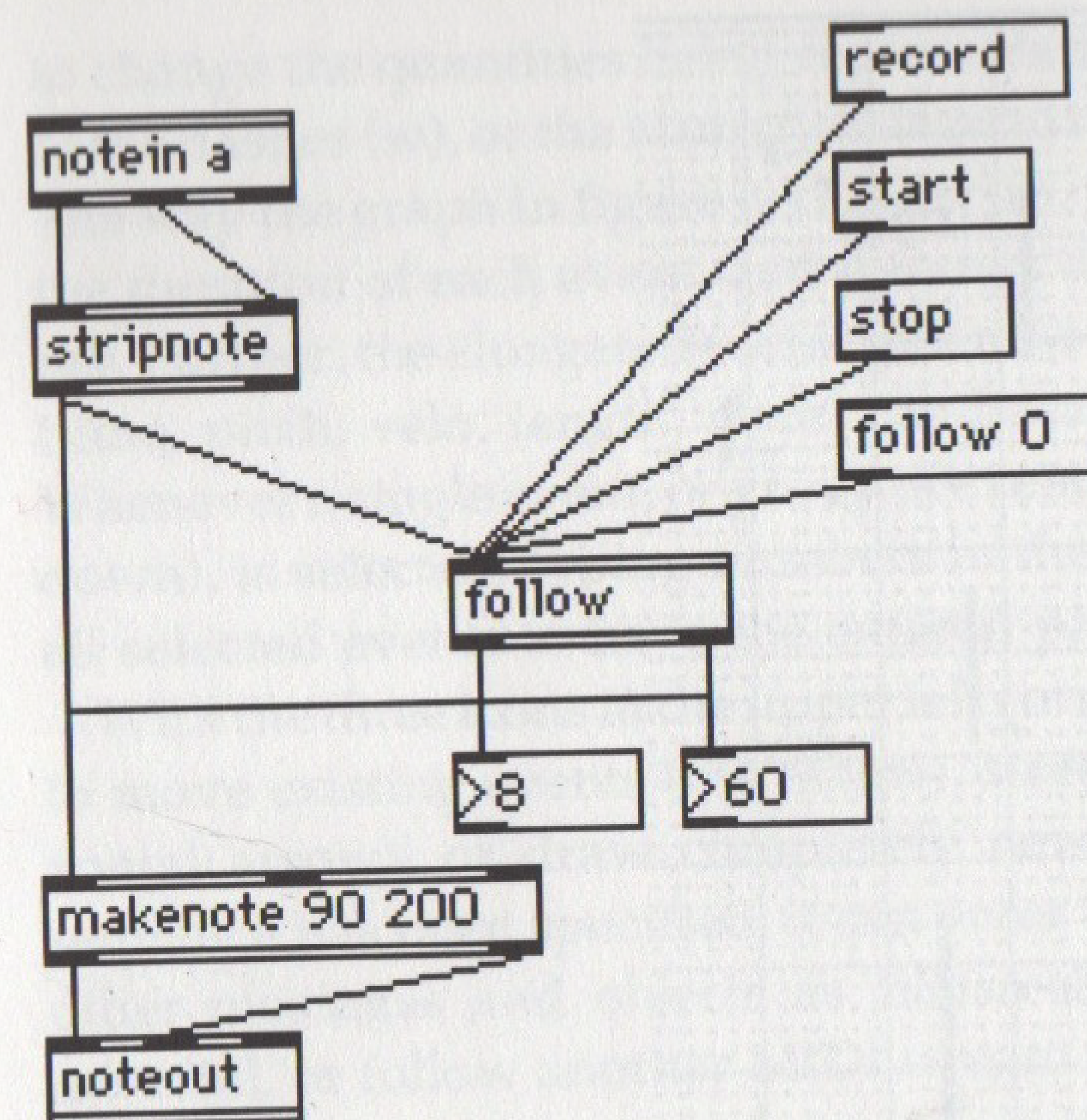

Figure 3.11

number of message boxes, lined up in the upper right corner of the patch. Clicking on the `record` message will cause *follow* to start recording the pitches arriving from *stripnote*. Then, clicking on `start` will make *follow* play the pitches back. The note numbers are sent out *follow*'s right outlet, where we can see them in the number box and hear them through *makenote/noteout*. Clicking `stop` will halt either recording or playback. Finally, once a sequence of note numbers is stored in the *follow* object, clicking on `follow 0` will start the object looking for the pitches of the sequence to be played again from the beginning. Each match between the stored sequence and new performance causes the index number of the matched pitch in the sequence to be sent out the left outlet; we can track the progress of the match with the number box attached to that outlet. Users can experiment with the leniency of the matching algorithm: playing some extra notes will not affect the match of the sequence when it is resumed. Skipping a note or two will cause the matcher to jump ahead as well. More radical deviations from the stored version, however, will eventually make the machine lose its place.

Explode is a graphic score editor built on top of the basic functionality of *follow*. Figure 3.12 shows an *explode* window: the black dashes in the main grid represent MIDI note events, where the *x* axis represents time, and the *y* axis note number. The chart above the grid makes it possible

3.4 Hyperinstruments

The *hyperinstruments* project, led by Tod Machover and Joseph Chung, has been carried out in the Music and Cognition group of the MIT Media Laboratory. "Our approach emphasizes the concept of 'instrument,' and pays close attention to the learnability, perfectibility, and repeatability of redefined playing technique, as well as to the conceptual simplicity of performing models in an attempt to optimize the learning curve for professional musicians" (Machover and Chung 1989, 186). Hyperinstruments are used in notated compositions, where an important structural element is the changing pattern of relationships between acoustic and computer instruments. As the preceding quote indicates, a focal point in developing hyperinstruments for any particular composition has been to provide virtuoso performers with controllable means of amplifying their gestures. Amplification is here used not in the sense of providing greater volume, but in suggesting a coherent extension of instrumental playing technique, such that a trained concert musician can learn sophisticated ways of controlling a computerized counterpart to his acoustic sound. One important expression of this concept has been to provide sophisticated control over a continuous modification of the synthetic timbres used to articulate hyperinstrument output.

Hyperinstruments are realized with the Hyperlisp programming environment, developed by Joseph Chung. Hyperlisp is available on the companion CD-ROM. Based on Macintosh Common Lisp Second Edition and the Common Lisp Object System (CLOS), Hyperlisp combines the power and flexibility of Lisp with a number of extensions designed to accommodate the scheduling and hardware interfacing needs of musical applications. Most of this functionality is built into an extensive library of objects, which have proved general enough to realize several quite different compositions, as we shall see shortly.

As the name would suggest, hyperinstruments follow an instrument paradigm. Applications are usually score driven, and the generation technique can best be described as a hybrid of generative and sequenced styles. Hyperinstrument compositions can be thought of as score driven, even though full-blown score following is generally not employed. The music to be played by the human performers is notated quite precisely, allowing latitude in expressive performance but not, generally, in the improvisation of notes and rhythms. Coordination with the computer is accomplished through a cueing system, where cues are sometimes taken from the MIDI representation of a performance and sometimes from the intervention of a computer operator.

Valis

The first composition to make use of the Hyperlisp/hyperinstruments model is *Valis,* an opera based on the novel of the same name by Phillip K. Dick. The music of the opera, beyond the vocal parts, is realized by two hyperinstrument performers, one playing piano, the other mallet percussion, where both instruments send MIDI streams to the computer for further elaboration. Several instruments were developed for different sections of the composition; here much of the groundwork was laid for the kinds of interaction that were to be developed in later compositions.

For example, one instrument animates chords performed on the 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 by playing through the score, 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 matching what is being played on the keyboard; successful matches are used to move from one rhythmic sequence to another. In that way, the example is score-driven: an internal score is matched against performer input, and successful matches move the computer performance from one rhythmic sequence to the next. The generation is a hybrid of sequenced and generative techniques, since the stored rhythmic score is a sequence, but one that is generatively modified, in timbral presentation and rate of change from one sequence to the next, by the performance of the human.

Towards the Center

Towards the Center (1988–89) is scored for flute, clarinet, violin, violoncello, keyboard, percussion, and computer system. Of these, the strings and woodwinds are miked and slightly processed, while the keyboard and percussion parts are full-blown hyperinstruments. In this piece, the hyperinstruments sometimes form "double instruments," in which control over the sounding result is shared between the two performers.

taken over time. A number of objects were added to Hyperlisp to interpret various kinds of performance gestures, such as note attacks, tremoli, and bowing styles. These objects can be "mixed in" to hyperinstrument modes active at different parts of the piece; rather than having all gestures tracked at all times, gestures with significance for particular sections of the composition can be interpreted as needed. Because of the powerful tracking techniques developed for continuous control and the previously mentioned difficulties of determining the pitch of an audio signal in real time, the gestural information available from a performance of *Begin Again Again* . . . is virtually the inverse of the normal MIDI situation: continuous controls are well represented, whereas pitch and velocity are more approximate.

Necessity became a virtue as the piece was composed to take advantage of the depth and variety of continuous gestural information available. For example, one section of the piece uses an "articulation mapping" in which different signal processing techniques are applied to the live cello sound, according to the playing style being performed. Tremolo thus is coupled to flanging, pizzicato to echo, and bounced bows to spatialization and delays. Further, once a processing technique has been selected, parameters of the processing algorithm can be varied by other continuous controls. Once tremolo playing has called up the flanger, for instance, the depth of chorusing is controlled with finger pressure on the bow (Machover et al. 1991, 29–30).

3.5 Improvisation and Composition

Interactive systems have attracted the attention of several musicians because of their potential for improvisation. Improvised performances allow a computer performer (as well as the human performers) broad scope for autonomous activity. Indeed, the relative independence of interactive improvisational programs is one of their most notable distinguishing characteristics. The composer Daniel Scheidt articulates his interest in incorporating improvised performances thus: "In making the decision to allow the performer to improvise, I have relinquished aspects of compositional control in exchange for a more personal contribution by each musician. An improvising performer works within his or her own set of skills and abilities (rather than those defined by a score), and is able to offer the best of his or her musicianship. Once having become familiar with [an interactive] system's behavior, the performer is free to investigate its musical capabilities from a uniquely personal perspective" (Scheidt 1991, 13).

Further, improvisation demands highly developed capacities for making sense of the music, either by a human operator onstage or by the program itself. If a human is to coordinate the responses of the machine with an evolving musical context, a sophisticated interface is required to allow easy access to the appropriate controls. If the machine organizes its own response, the musical understanding of the program must be relatively advanced.

George Lewis

A strong example of the extended kinds of performance interactive systems make possible is found in the work of George Lewis, improviser, trombonist, and software developer. His systems can be regarded as compositions in their own right; they are able to play with no outside assistance. When collaboration is forthcoming, human or otherwise, Lewis's aesthetic requires that the program be influenced, rather than controlled, by the material arriving from outside. The system is purely generative, performance driven, and follows a player paradigm. There are no sequences or other precomposed fragments involved, and all output is derived from operations on stored elemental material, including 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's trombone. A listening section parses and observes 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 are always available to contribute to the calculations, but are only invoked some percentage of the time—this probability is set by the composer or 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.

George Lewis's software performers are designed to play in the context of improvised music. The intent is to build a separate, recog-

nizable personality that 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's 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, not adopted from the input. Further, the stylistic elements recognized by the listening section are made available to the generation routines in a way that elicits responsiveness but not subordination from the artificial performer.

Richard Teitelbaum

Richard Teitelbaum's *Digital Piano* collection is an influential example of a transformative, performance-driven, instrument paradigm interactive system. The setup includes an acoustic piano fitted with a MIDI interface, a computer equipped with several physical controllers, such as sliders and buttons, and some number of output devices, often including one or more solenoid-driven acoustic pianos. This particular configuration of devices arose from Teitelbaum's long experience with live electronic music, beginning with the improvisation ensemble Musica Elettronica Viva in 1967. Two motivations in particular stand out: (1) to maintain the unpredictable presence of human performers, and (2) to use an acoustic instrument as a controller, to contrast with the relatively static nature of much synthesized sound (Teitelbaum 1991).

Using the *Patch Control Language*, developed in collaboration with Mark Bernard (Teitelbaum 1984) (or, more recently, a reimplementation in Max of the same design), Teitelbaum is able to route musical data through a combination of transformations, including delays, transpositions, and repetitions. Before performance, the composer 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's parameter values and to use buttons or an alphanumeric keyboard to break or establish routing between modules. In this case, the musical intelligence employed during performance is 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 composer'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.

A new version of Patch Control Language, written in Max by Christopher Dobrian, has extended Teitelbaum's design. The *transposer* subpatch is shown in figure 3.19. Across the top are three inlets, which will receive pitch, velocity, and transposition values from the main patch. When the velocity value is nonzero (that is, the message is a Note On), the transposed pitch corresponding to the pitch played is saved in the *funbuff* array. Notes with a velocity of zero are not passed through the *gate* object and so call up the stored transposition: in this way, even if the transposition level changes between a Note On and Note Off, the correct pitch will be turned off.

Another module, the *excerpter*, is shown in figure 3.20. The *excerpter* is built around the sequencer object visible on the right-hand side of the patch, between *midiformat* and *midiparse*. The rightmost inlet, at the top of the patch, sends packed pitch and velocity pairs from MIDI Note On and Off messages to the sequencer for recording. The next inlet from the right, marked "playback tempo," supplies exactly that: scaled through the *split* and *expr* objects, this inlet eventually sets the argument of a start message. The next inlet to the left sets a transposition level for pitches played out of the sequencer and simultaneously fires the start message, which begins playback from the sequencer at the

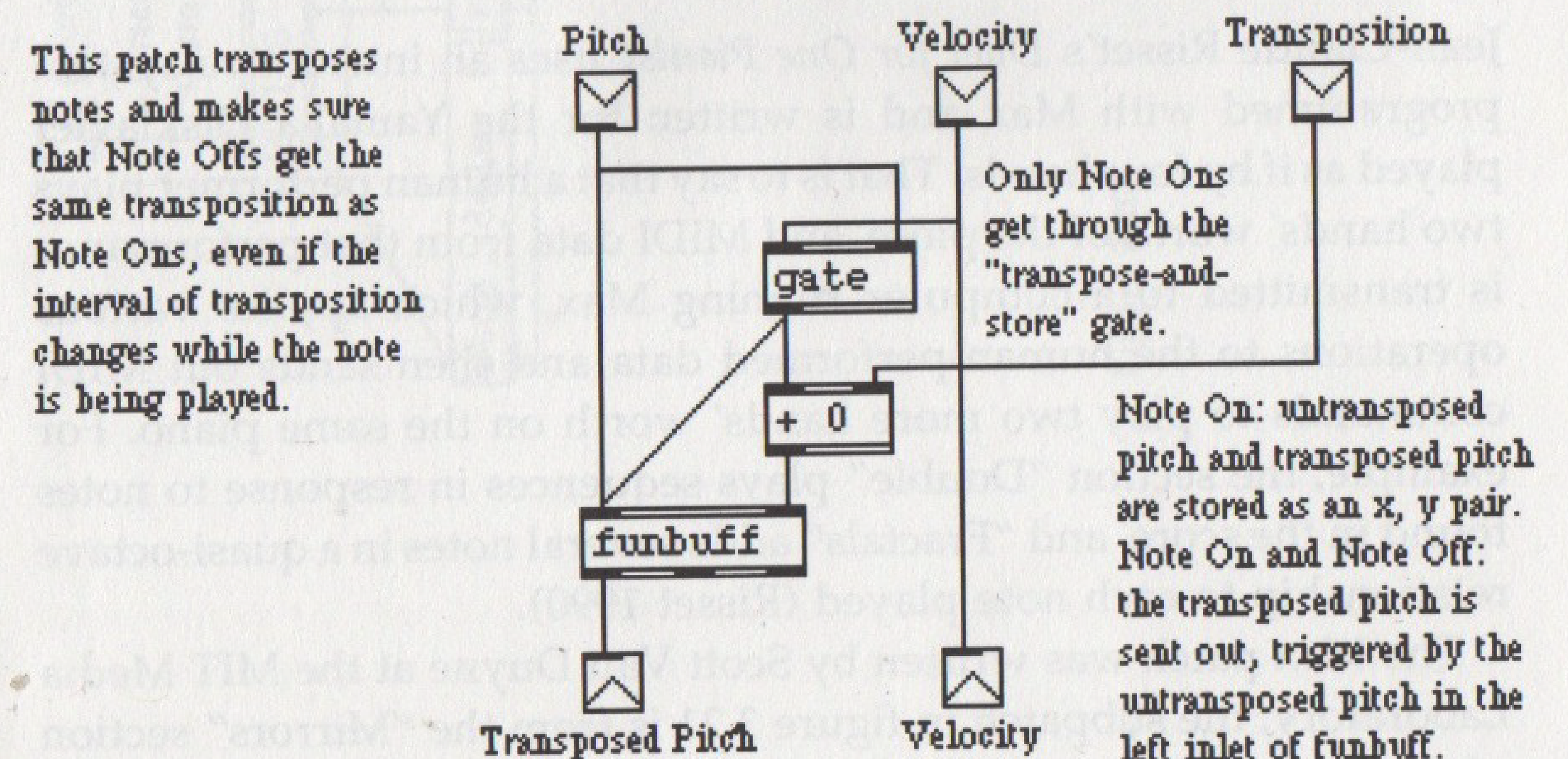

Figure 3.19

behavior of a computer program (Winograd and Flores 1986). The work described here actually exists and has been used with human musicians in a wide variety of settings over a period of several years. Programs are said to "understand" or exhibit "intelligence" in the sense that musicians interacting with them feel that the programs are behaving musically, that they can understand what the systems do, and that they can see a musically commonsense relation between their performance and the program's. The word "understanding" is used to describe a quality of this interaction as it is experienced by musicians performing with a program.

4.2 Hierarchies and Music Theory

In this section I present some structural perspectives developed in the field of music theory, particularly those related to questions of hierarchy and progression. The point of the discussion is to gauge the descriptive power of each organizational scheme, 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 theories of music.

Heinrich Schenker

In the early part of this century the Galician music theorist Heinrich Schenker developed an analytic technique with significant implications for the questions we are considering here. In particular, his work explored the combination of a strong concept of structural levels, with the idea of progression from event to event within levels. The hierarchical nature at the basis of Schenker's thought is readily apparent: "[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 Schenker 1933, 14).

In the Schenkerian analysis shown in figure 4.1, notice the reduction of musical surface structures to a series of nested levels, with directed motion between certain events within each level. Such a graph portrays a double perspective on the musical flow: hierarchical in its layered organization and progressive in the implications between events within levels.

F. CHOPIN ETUDE IN F MAJOR, OP. 10, NO. 8 Hintergrund und Mittelgrund

Figure 4.1 Reprinted from Heinrich Schenker, Fine Graphic

Beyond the use of hierarchy and progression, another contribution from Schenker's theory is his use of *recursion*. Recursive techniques use the same process on distinct levels of a problem, telescoping up or down to find similar structures on varying scales. "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" (Cook 1987, 36).

These three ideas—hierarchy, 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 that 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. "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?" (Schenker 1979, 162; see also Narmour 1977, 38).

Schenker's rejection of music that did not fit 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 strongly 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.

We may conclude by reiterating some relevant contributions of Schenkerian analysis—structural levels, progressive motion within levels, and recursion—and the cautionary tale of its inventor's application of them: 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 illuminate.

Lerdahl and 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 and Jackendoff 1983). The combination of Chomsky and Schenker is certainly not incidental: as noted by John Sloboda, "their theories have some striking similarities. They both argue, for their own subject matter, that human behaviour *must* be supported by the ability to form abstract underlying representations" (Sloboda 1985, 11).

Lerdahl and Jackendoff's theory is designed to produce representations of pieces of tonal music that correspond to the cognitive structure present in an experienced listener's mind after hearing a piece. "[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 and Jackendoff 1983, 6).

They focus on those parts of music cognition they consider to be hierarchical in nature.

> We propose four such components, all of which enter into the structural description of a piece. As an initial overview we may say that *grouping structure* expresses a hierarchical segmentation of the piece into motives, phrases, and section. *Metrical structure* expresses the intuition that the events of the piece are related to a regular alternation of strong and weak beats at a number of hierarchical levels. *Time-span reduction* assigns to the pitches of the piece a hierarchy of "structural importance" with respect to their position in grouping and metrical structure. *Prolongational reduction* assigns to the pitches a hierarchy that expresses harmonic and melodic tension and relaxation, continuity and progression. (Lerdahl and Jackendoff 1983, 8–9)

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 and Jackendoff's work is that it treats musical rhythm much more explicitly than do many music theories, Schenker's being an example. In fact, they regard their theory as a foundation for Schenker's work: where his graphs highlight the relative importance of different pitches in a composition, Lerdahl and