M and Jam Factory

David Zicarelli
Intelligent Music
P.O. Box 8748
Albany, New York
12208 USA
and
Center for Computer Research in Music and
Acoustics (CCRMA)
Stanford University
Stanford, California 94305 USA

Introduction

M (Fig. 1) and Jam Factory (Fig. 2) are the first software packages published by Intelligent Music, a company founded by composer Joel Chadabe to provide a commercial outlet for interactive composing software (Chadabe 1984) and intelligent instruments. M and Jam Factory, commercially available since December 1986, run on Apple Macintosh computers equipped with a MIDI interface (Chadabe and Zicarelli 1986, 1987).

Both M and Jam Factory are programs that, through the use of graphic and gestural interfaces, provide an environment for composing and performing with MIDI. The resulting environment, characterized by controls that offer immediate feedback in modifying an ongoing process, is quite analogous to the control panel of an airplane. Indeed, one of the early models for the programs was Fokker Triplane, a flight simulation program for the Macintosh.

The original design for M was developed by Joel Chadabe, John Offenhartz, Antony Widoff, and me in early 1986. M is a specific embodiment of the kind of interactive composing system discussed by Chadabe, but the program has grown into a laboratory for generating and processing MIDI far more complex and powerful than its original conception.

Jam Factory is based on an algorithm that employs transition tables, also known as Markov chains. The motivation for writing Jam Factory was my interest in creating a program that would listen to MIDI input and "improvise" immediately at some level of proficiency, while allowing me to improve its ability. This could be done by playing more notes or by adjusting the parameters of the algorithmic process while hearing the results of the process. Although it is clear that rules about musical practice (such as how to improvise) tend to limit the applicability of the algorithm to the aesthetics of the rulemaker, I hoped, by providing a wide and flexible range of controls, to give the user of Jam Factory the opportunity to personalize the results produced by the program.

The rest of this article provides an overview of the current state of the development of both programs, and explains their architectures and some of the algorithms they employ.

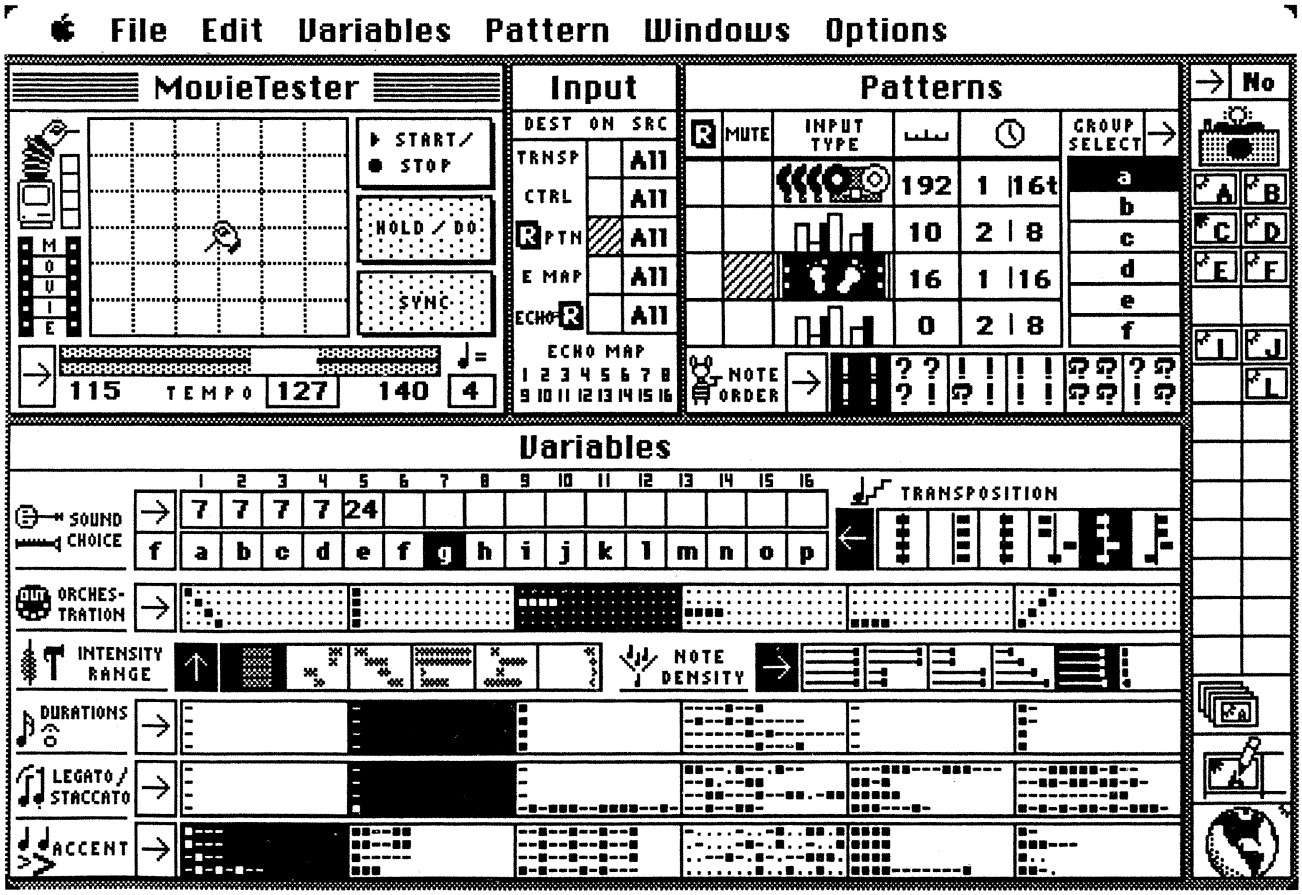
Computer Music Journal, Vol. 11, No. 4, Winter 1987,

A Style of Interactive Composing

Chadabe (1984) describes a process of composing (which he calls "Design then do") where the composer uses some tools (a programming language or an algorithm design application) to design a "composing machine," which has some elements that are determined by user gestures. The next phase of use is to test the machine by performing the gestures. Usually the gestures adjust parameters or select different musical materials.

In M and Jam Factory, the distinction between the act of making the machine and using it are blurred. The programs themselves are too general to be considered compositions per se, and it is entirely possible to modify the changeable parts of the "machine" in the "performance" such that you are performing another machine. At certain times, the user is in the design stage, at others the user is in the performance stage. Because the program's musical output immediately reflects a gesture made to

Fig. 1. Main screen for M.



something on the screen, many users can build a composition as a performance. M and Jam Factory design activities can be viewed as performance gestures: the range bar (discussed later) is a primary example of this. Users can impose arbitrary distinctions between composing and performing by restricting their actions at specific times.

The User Interface

The user interface for M and Jam Factory consists of a number of objects arranged as a graphic control panel drawn on the Macintosh screen. The control panel objects are manipulated (with a mouse, for example) to change parameters of an automatic musical process while that process runs, making music. A few characteristics of such objects include the following:

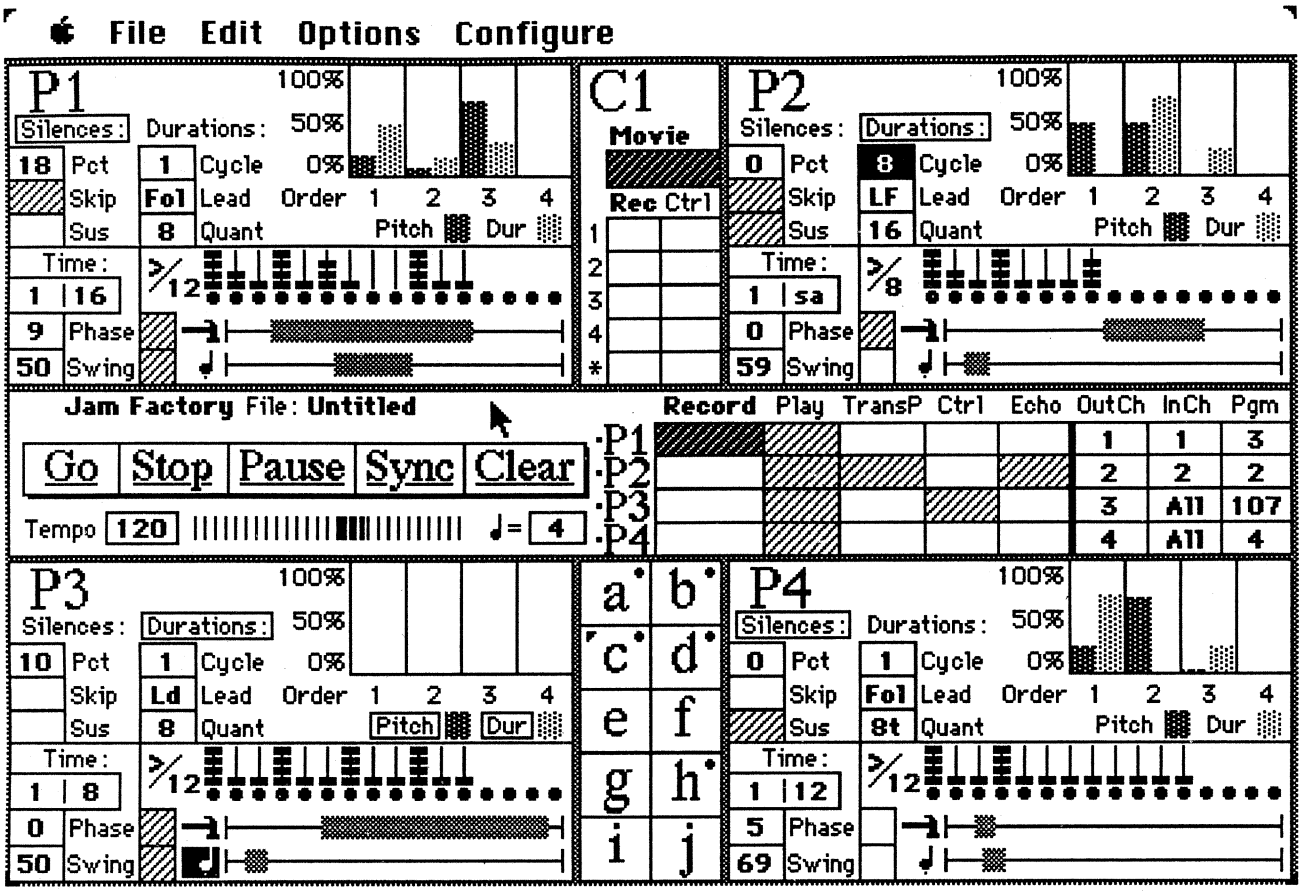
Each object communicates the current value of a musical process parameter and imparts some knowledge of what would happen if the parameter were changed. It is important that the program communicate to the user a range of possible courses of action to take, as well as allowing those actions to take place easily.

Each object presents a musically powerful control that makes a perceived difference when its value is changed.

Specific control settings, and consequently specific states of the musical process, can be captured and retrieved.

The graphic and operational nature of the objects themselves was extremely important in the design, and we found that the normal Macintosh Toolbox objects (such as scroll bars) were not suitable for an interactive music application. We developed a new

Fig. 2. Main screen for Jam Factory.



set of tools that operate smoothly, take up less precious Macintosh screen real estate, and, most important, effectively communicate notions of algorithmic composition. The objects are implemented such that they can be scaled to any desired size and can take on a variety of appearances. They include *range bars, numericals, choice bars,* and *button matrices* (Fig. 3).

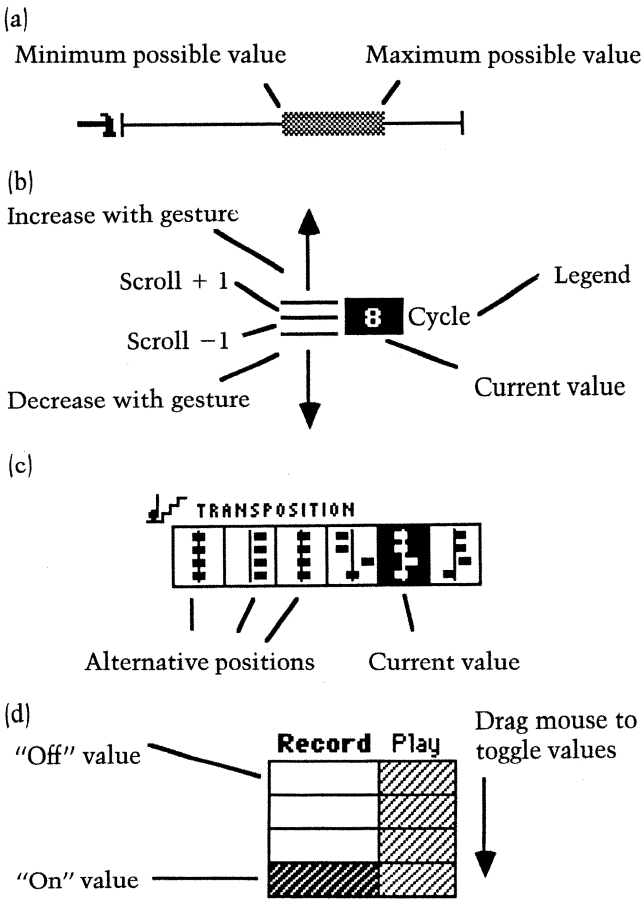
The range bar is a simple and elegant way of allowing a user to describe a range of values while succinctly communicating the idea of controlled automatic variation. The user "draws out" the desired range by dragging with the mouse from one end of the bar to the other. In many cases, numerical displays (to show the values of both ends of the range) are changed as the mouse is being dragged.

The numerical is a device for displaying and changing information represented as a number or as one of a small number of strings (such as "True," "False," or "Either Is Fine With Me"). The numerical usually is contained within a box that is drawn as part of a window's background picture. The value to be changed appears in bold, and there is usually a legend indicating what it is that the number represents, which is not bold. Several methods can be used to change the value of a numerical:

Holding the mouse button down in the top half of the box scrolls the value up by 1, and holding the mouse button down in the bottom half scrolls the value down by 1.

Dragging the mouse button above or below the outline of the box scrolls the value in direct proportion to the amount of mouse movement. Since it disappears, and since its motion is not confined by the screen boundaries, the mouse can move "forever."

Fig. 3. Four of the choice facilities used in M and Jam Factory. (a) range bar. (b) numerical. (c) choice bar. (d) button matrix.



In Jam Factory, the numerical can be selected and the value typed in with the Macintosh keyboard or, in some cases where a note is to be requested, played on a MIDI keyboard.

Although it is not always the most suitable solution, the numerical can solve just about every user interface problem. It doesn't take up much screen space, and it offers the manipulation and gestural feedback possibilities of a scroll bar. I devised the numerical as a control in my DX/TX Patch Editor, published by Opcode Systems, and it has been adopted in a number of programs since that time.

The choice bar is used to represent the selection of one out of a set of uniformly sized objects. I used it to represent M's variables, each of which has six possible values. To change a choice bar position, the user clicks on the desired position. The structure of a choice bar also makes it easy to drag one position onto another, for purposes of copying or swapping values. A choice bar, of sorts, that every Macintosh user has seen is the palette of tools in the MacPaint program.

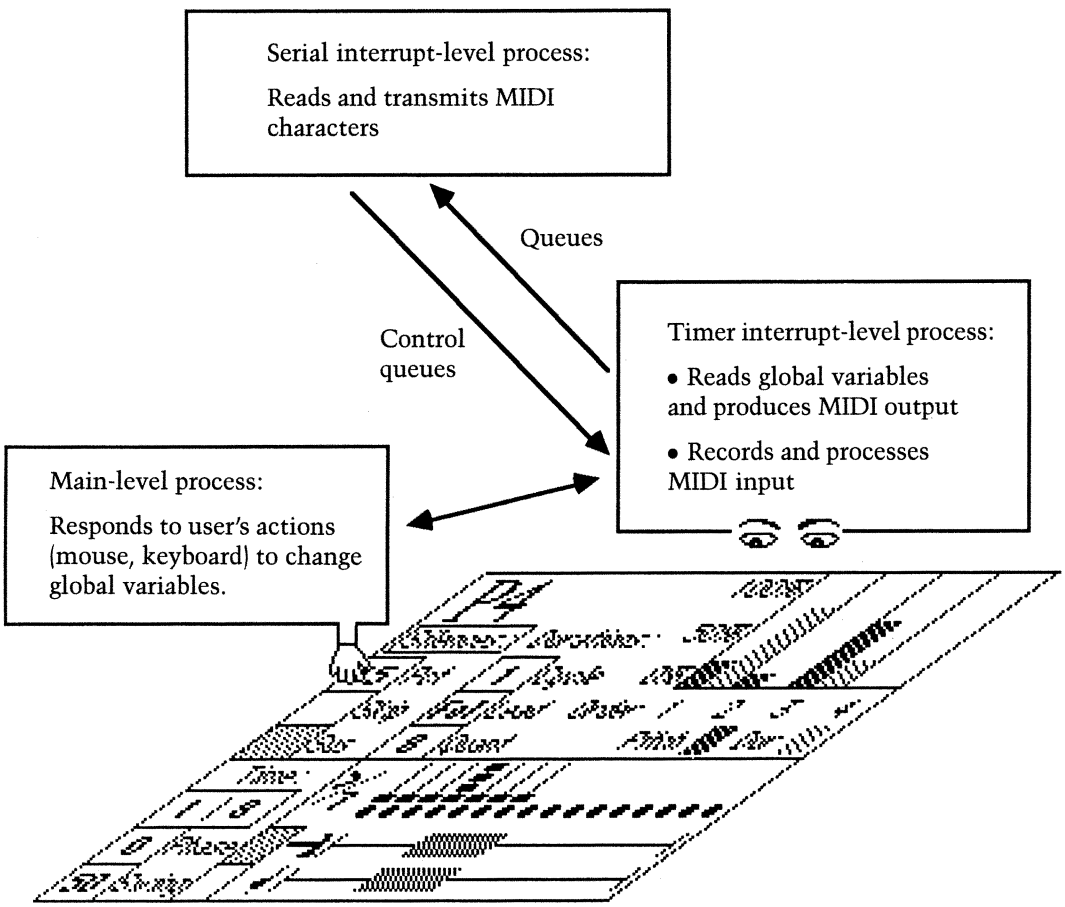
The button matrix is a way of representing an on-off state for a set of items. The "on" state is represented by a graphic "pattern" chosen by the program designer, and the "off" state is shown as white.

The Implementation Architecture

In general, the Macintosh provides a very friendly environment for the development of interactive music programs. The computer is reasonably fast and the screen is sharp enough to create a user interface of substantial detail and complexity. More importantly, the architecture of the Macintosh is set up to facilitate MIDI communication and high-resolution timing of events. (We are obliged to Dave Oppenheim, author of the Opcode Sequencer, for his work in developing the MIDI and timing routines we have used.) To send and receive characters, the MIDI driver uses serial chip interrupts that have a higher priority than the millisecond interrupts used for the computation of outgoing notes and processing of incoming notes. The millisecond interrupts are used in this way so that the program can play music and pay attention to the user at the same time. The various levels of operation are detailed in Fig. 4.

Communication between levels is done through the use of message queues. The play and MIDI input handling routines use queues to request output (of a MIDI note, for example) or buffer input from the MIDI driver. A control queue is used to pass messages between the timer interrupt level and the main level. Since the processing of incoming MIDI data occurs at the interrupt level, one extensive use of the control queue is to update the screen in response to MIDI control keys (called the input control system) or other MIDI notes which are received by the interrupt routine. Communication from the main loop to the interrupt is necessary in cases such as hitting the stop or start button or some user-triggered transmission of MIDI data (such as typing in a program change).

Fig. 4. Relationship of the processes in M and Jam Factory to facilitate interactive composition. The process activated in response to a serial interrupt transmits and receives individual characters of MIDI data. The process activated in response to a timer interrupt (shown by the eyes) observes the state of the musical global variables and calculates output or processes input accordingly. The main-level process (shown by the hand), which does not run at interrupt level, allows the user to change the musical global variables, which in turn affect the calculations of the timer-interrupt process.



The major method of communication between levels, however, is that the user interface acts on the same set of global data that the interrupt process uses to compute notes. When the mouse is used to change a parameter from 9 to 10, for example, the next time that parameter is used to compute a note, the change will be taken into account and heard. This seemingly simple idea is at the heart of the technique of writing programs that play music that changes according to user input.

In writing the code at the interrupt level, the goal has been to leave things in such a state that the computation of notes can be done in as little time as possible. In particular, a great deal of time has been spent eliminating costly 68000 multiply and divide instructions. For example, many commonly used ranges of pseudorandom numbers (such as those between 0 and 100 for use in random percentages) are computed when the program starts up and read out from a cyclic table.

Two routines run at interrupt level. The tick routine services the data that needs to go out. It executes once every 96th of a quarter-note, with the actual number of milliseconds between ticks set by the tempo control. The record routine, which executes every 8 msec, services the queue of MIDI input.

The basic tasks done by the tick routine are as follows:

It increments the number of ticks since the user hit the start button.
It checks to see if any note-offs are scheduled to be played, and if so, it plays them.

For each pattern (M) or player (Jam Factory), it decrements the time-base denominator and/or numerator counters, and if the numerator is 0, it executes the note-playing routine and resets the counters.

It checks if it is time to click the speaker for the metronome or send out a MIDI timing signal.

It handles any requests for sending MIDI data, such as program changes or other MIDI commands.

The record routine checks to see if there is any data waiting in the input queue, and if so it does the following:

If the data is MIDI timing information, it updates the synchronization counters and makes adjustments to the tempo.

If the data is being echoed to the output port, it "rechannelizes" the data and sends it out.

If the data is not a note or program change, it's generally ignored (M has a "sequencer" mode where this information is recorded).

It executes any recording, keyboard transpose, or input control routines that may be enabled and applies them to the data. Some of these routines may place things in the control queue to request updating of the screen or other activities that cannot be performed in the interrupt.

The main loop of the program is a typical Macintosh endless "event loop," which checks if there's anything that an interrupt routine has requested via the control queue, checks for running out of memory if it is capturing a performance (we call the capture of a performance a *movie*), and then handles any mouse, keyboard, or operating system events.

The Algorithms

In designing M and Jam Factory, various assumptions—effectively musical decisions—were made in order to promote a user's ability to "get at" the music and change it interactively while maintaining significant control.

First, changing the pitches in both programs consists primarily of reordering. M uses a different reordering scheme than the transition tables of Jam Factory, but in essence both programs achieve similar results. Reordering is a powerful way of thinking about interactive control of pitch for the following reasons:

It is extremely easy to make small changes in the parameters of the reordering algorithm to adjust the output to something more desirable. For example, in Jam Factory, you can click on a picture of a bar graph to change the relative distribution of the orders of the transition tables used. The perceptual effect of this is usually very obvious, and given that you hear the changes instantly, it is not hard to find a setting that is suitable for the material. Instead of having the computer work hard to come up with a variation appropriate to the style the composer has chosen to work in, the program gives a composer the ability to supply the essential information that can help make it appropriate.

The user has control over the tonality of the piece by supplying the pitches to be reordered. Most composers would probably agree that it is of higher importance what notes are played than what order they're played in, although this is a conjecture which should not (and does not) necessarily dominate the design of every algorithmic composing program.

Yavelow (1987) points out that subtle reordering, such as the kind possible with transition tables, can be a substitute for direct repetition (or *looping*) of a phrase.

Second, control over rhythm is approached in both programs on two levels. A master tempo, expressed in beats per minute, is translated into the amount of time for each tick, the smallest unit of time resolution. The *time base* modifies the tempo independently for each musical voice by expressing the number of ticks between events. The time base is represented as a rational expression, where the numerator ranges from 1 to 99 and the denominator is a fraction of a whole note selected from a list including 1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 12, 13, 15, 16, and 24. The denominator translates into a number of ticks (for instance, a denominator of 4 = 96 ticks).

Step advance is a special denominator where the user plays the durations on a MIDI keyboard.

Each program has its own method of handling durations above this level, but the effective duration will be a simple multiple of the time base. Jam Factory allows further control over what could be termed "micro timing." *Swing* creates uneven eighth notes, and *time distortion* creates rubato over an arbitrary time period. These two features do not affect the overall speed of a process as would changing the time base.

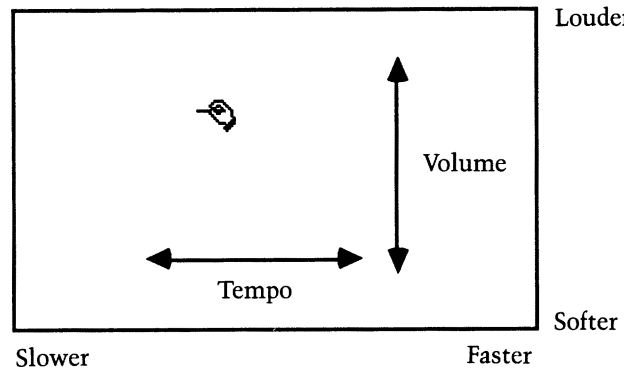


Fig. 5. The original conception of conducting in M used the mouse to change parameters in a continuous fashion based on its position in two dimensions. "How can we make a three-dimensional mouse?" we asked.

A General Description of M

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. These parameters, in M, are called *variables.*

Patterns are classified by input type (the way that the notes of the pattern are entered) as follows:

Distribution patterns contain single notes and can be entered from a MIDI keyboard or by clicking on a *skyline* keyboard in a pattern editing window. The notes placed in two lists—the original list and the scrambled list (which is a randomly reordered version of the original list). The pattern ordering variable chooses a note from one of the two lists (or just randomly from the original list) using a distribution of percentages.

Step record patterns are identical to distribution patterns except that chords can be entered (using a chord-detection algorithm) and the input must come from a MIDI keyboard.

Drum machine patterns are either distribution or step record patterns which are initially filled with rests. Recording takes place in real time from a MIDI keyboard and new notes replace rests. Reordering is done as follows: a third list contains the locations of all the nonrests in the original list, and creates the scrambled list by swapping these locations. Thus, there isn't a reordering of the positions of rests and notes.

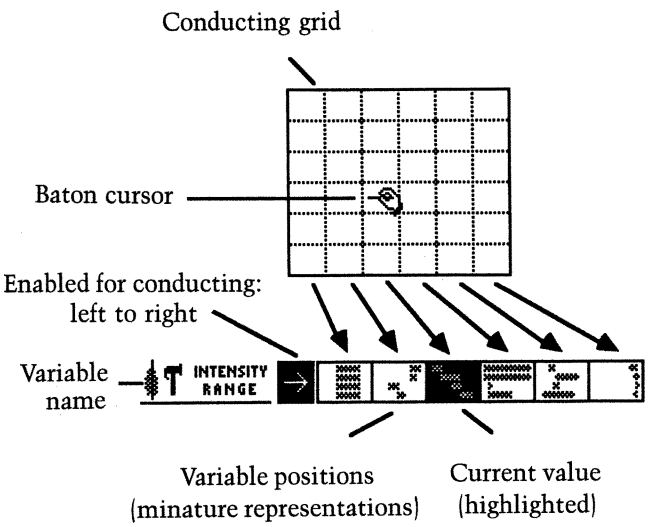
Real-time patterns are like traditional sequencer patterns, and bypass most of the M variables. They have an adjustable length, can be quantized in real time, and can be speeded up or slowed down relative to the master tempo in certain basic ways.

M provides facilities for manipulating pattern information, including copying, pasting, filtering, and editing.

At the beginning of program development, M was oriented around a large area in which the mouse would be used to change variable values. The concept is illustrated in Fig. 5. But as work progressed, we realized that this notion of conducting, in which the mouse gesture was encouraged to be continuous, was not appropriate for all kinds of changes that were made in performance. We found, for example, that timbre selection (sending MIDI program changes) was far better controlled with a discrete set of alternatives than in a continuum.

Each M variable has six positions, arranged in sequential order, which contain alternative configurations of a particular parameter. Miniature representations of each of the six positions for all the variables are displayed on the main screen at the same time. One of these positions is highlighted, showing the current value of the variable. To switch between the positions of a variable, the user merely clicks the mouse on the desired miniature representation (which imparts predictive knowledge about the action-to-be). To edit the settings of one or more variable positions, the user brings up an editing window that displays the information in a detailed textual and/or graphic form. If M is playing, and the selected position of a variable is edited, the changes are reflected immediately in the musical output.

Fig. 6. Current conducting setup in M tracks the changing of discrete alternatives in variables with the position of the mouse in a grid. The user can conduct a variable in one of four "directions," which associate grid locations with variable positions. Here, the left-to-right direction is shown.



Thus, the changing of the variables can happen both incrementally and interactively, and can be viewed as another kind of performance gesture besides that initially foreseen with the use of a conductor feature.

The six positions of each variable correspond to the six units along either axis of the conducting area where mouse gestures are made, as shown in Fig. 6. Conducting variables automates the selection of variable positions by mapping the mouse location in the grid to a variable position. Next to each variable is a conducting arrow, indicating the direction of movement of the mouse with respect to the variable's current value. If the arrow is highlighted, the variable is enabled for conducting, and the position will change in concert with mouse gestures in the conducting area. Any number of variables can be conducted simultaneously. Conducting thus becomes a way of selecting between discrete choices, rather than continuous modification of a parameter. The lone exception to this is tempo: conducting changes the tempo within a range drawn out by the user. The tempo range is scaled to equal the size of the conducting grid.

M consists of four patterns, which sound simultaneously, and each pattern contains a separate value at each variable position. For example, Fig. 7 shows two positions of the note density variable, which controls the percentage of random "skips" that occur in playing a pattern.

In Fig. 7, position 1 can be interpreted as follows: pattern 1 plays all the time, patterns 2 and 3 play very rarely, and pattern 4 doesn't play at all. Thus switching to position 1 of the note density variable causes pattern 1 to play much more than all the other patterns. Switching to position 2 effectively shuts off all patterns except pattern 4.

A special variable called a *pattern group* contains the musical material that the four patterns play. Each pattern in a group could be considered a kind of track in a typical MIDI sequencer, but it is really more of a collection of notes or chords. The M algorithm attempts to "animate" these collections into music, adding rhythm, articulation, and dynamics.

Each variable has a miniature representation on the main screen. We've tried to give each type of variable its own graphic identity, and continue that identity into the variable editing windows. Each "row" in a miniature representation or editing window represents the information for each pattern.

The *orchestration* variable (Fig. 8) assigns patterns to MIDI channels (usually different synthesizers). It takes the form of a 4-by-16 matrix in which each of the four patterns can be assigned to any or all or none of the 16 MIDI channels.

The *sound choice* variable actually contains 16 arrays; in each are stored sets of MIDI program changes which change the timbre of a synthesizer. These program changes are sent every time a new sound choice alternative is selected.

The *note density* variable is the percentage of time that notes of each pattern are played. When the value is less than 100, a probabilistic algorithm decides whether to skip a note based on the percentage.

The *transposition* variable contains an offset, expressed as a note above or below middle C, which is added (or subtracted) to the note values of each pattern.

A *cyclic distribution* is a data structure that has some number of scalar values or random ranges that are read out once each tick of a clock. In M, these structures are applied to duration, articulation (le-

Fig. 7. Miniature representation and edit window display of two note-density-variable positions.

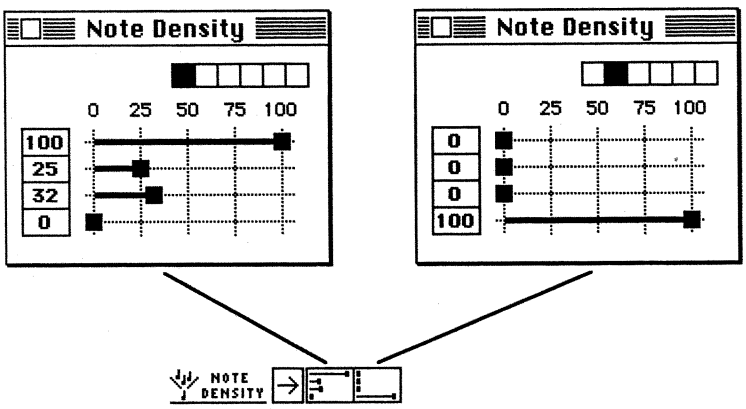


Fig. 8. Edit window of the orchestration variable.

Fig. 8

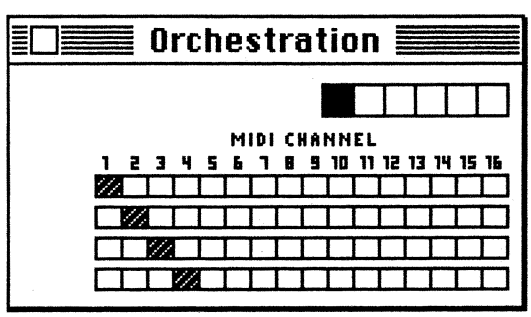
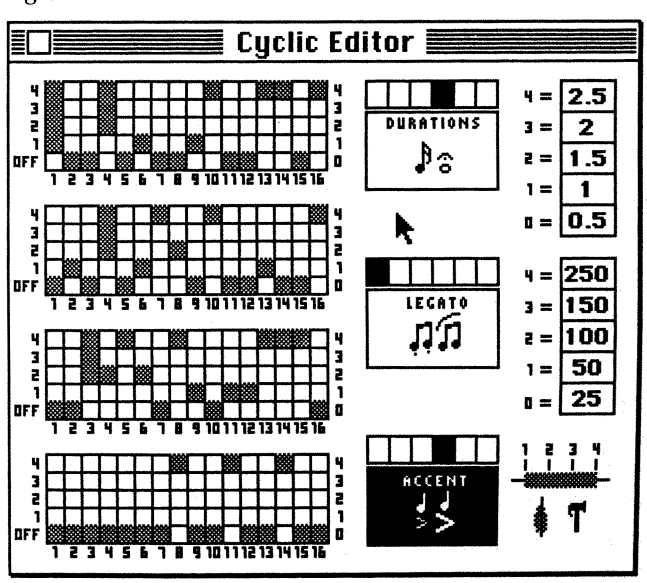


Fig. 9. The Cyclic Editor allows interactive editing of cyclic distributions applied to duration patterns, articulation (called legato-staccato), and velocity (called accent). The window displays one position of one of these variables which consists of four distributions, one for each pattern. The user can bring up for editing one of the six positions for any of the three variables on the right hand side of the window. The column of numericals on the far right translate the levels of the durations and legato-staccato variables into actual values. The editing of the distributions takes place on the left side of the window.

Fig. 9



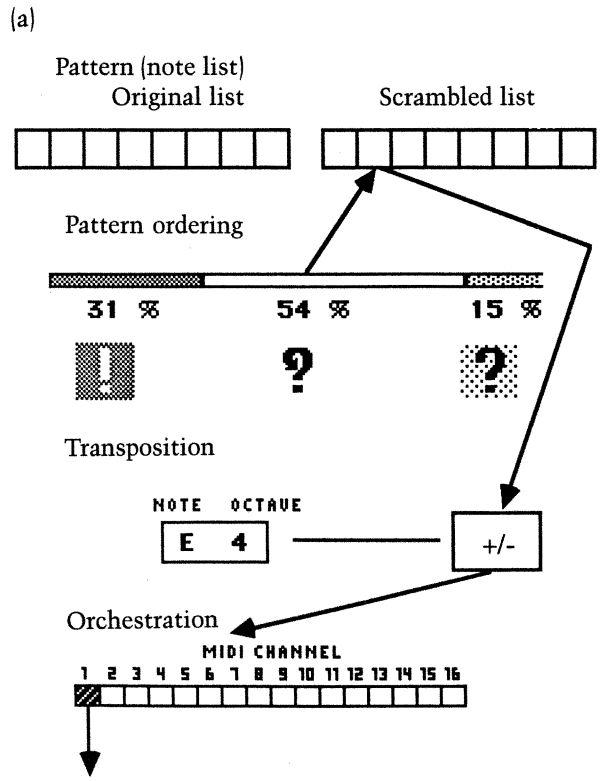
gato versus staccato), and accent (MIDI velocity). The implementation of these structures is quite simple, but they provide an elegant method of controlling variation through randomness. Figure 9 shows an editing window for a cyclic distribution. Time is horizontal, value (called level) is vertical. The levels can range from 0 to 4. Each level is defined in terms of an absolute value; for example, the levels for Legato-Staccato are mapped into percentages chosen by the user. A level of 0 might be 10 percent of the time between the beginning of the note and the beginning of the next note; a level of 4 might be 300 percent (in which case the notes will overlap). The levels don't need to be assigned to values in ascending order, so it's possible to weight the randomness by assigning the same value to more than one level.

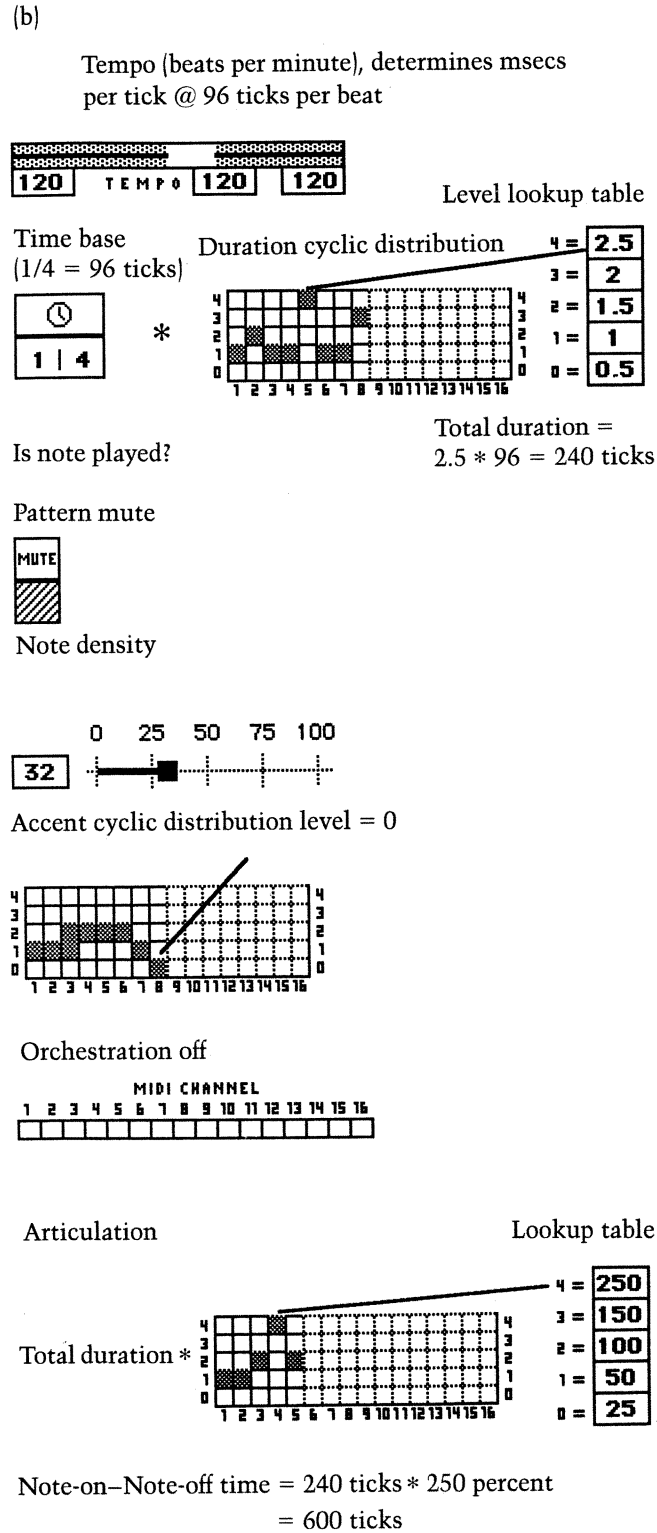
One position of a cyclic distribution variable contains distributions for four patterns. Each distribution has a unique length, so patterns cycle through their distributions independently.

The *Cyclic Editor* (shown in Fig. 9) shows the distributions for each of the four patterns. The length of the solid grid shows the length of the cycle, and the gray squares indicate the levels. Levels that extend over more than one square indicate random ranges.

These variables work together as the parameters of the M algorithm. The algorithm's computation of pitch, timing, and velocity is charted in Fig. 10.

Fig. 10. The interaction of M variables in the computation of pitch, timing, and MIDI velocity information. (a) Pitch computation. (b) Timing computation. (c) Velocity computation.





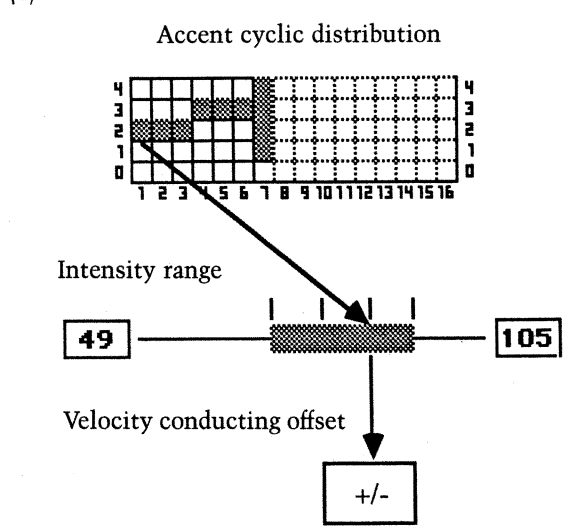
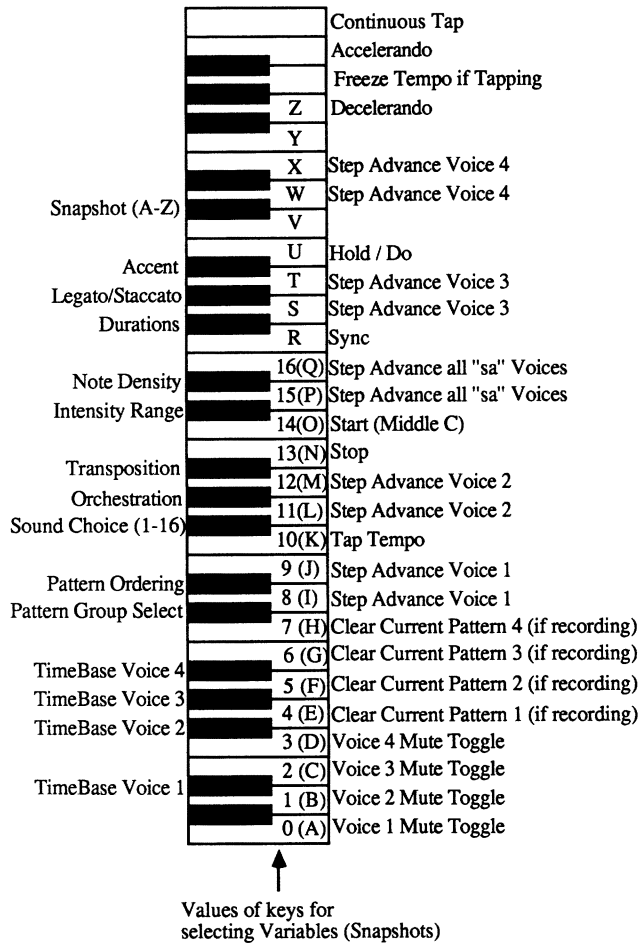


Fig. 11. Assignments made to each key on the keyboard for the input control system.



Settings of variables can be grouped as a *snapshot*, which can be recalled to select those settings simultaneously. A snapshot is made by clicking on the hold/do button (which begins to blink), then changing the positions of variables that are to be grouped together, each of which starts to blink when selected. Then the desired location (there are 26, labeled A–Z) for the snapshot is selected in the snapshot window, which stops all the blinking and creates the snapshot. The purpose of the snapshot is to enhance the gestural capability of the user, which is otherwise limited to either the changing of variable positions that are linked in position by conducting, or selecting variable positions one at a time directly with the mouse. The snapshot can be used to create sections of a piece delineated with combinations of sounds or musical material.

In both M and Jam Factory a control mode called the *input control system* is used to allow MIDI input to trigger certain program actions. Figure 11 shows the assignment of keys to various input control functions in M. The music can be stopped and started, variable positions can be changed, patterns muted, and so forth. Three other features are more gestural in nature:

Step advance triggers the choosing and playing of the next event in one or more patterns. The note is played when the performer depresses one of the assigned step advance keys, and is turned off when the key is released. The note is also played with the velocity that the key was depressed with. Step advance keys are grouped as two contiguous white notes so that successive notes can be played rapidly using a two-fingered technique. Each pattern has its own step advance keys, so any number of patterns can be performed in this manner, theoretically by more than one person.

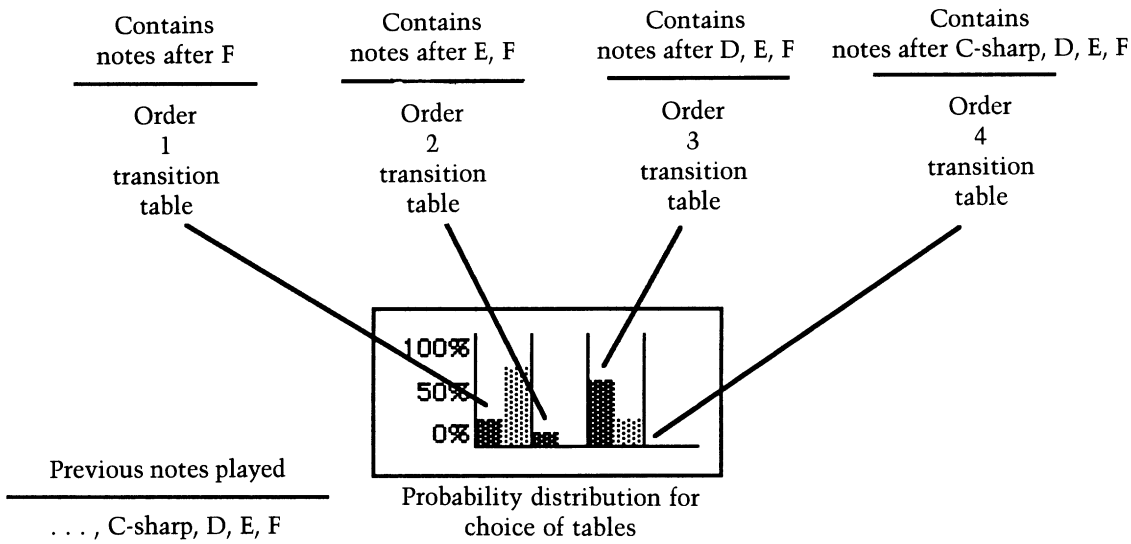
Tap tempo allows the tempo to be set from a MIDI keyboard: the time between two successive taps of the assigned key is measured and taken to be the time of a quarter-note.

Continuous tap tempo (for lack of a better word) follows the performer precisely in tapping out a tempo. Although the system does not try to pick up the tempo if no tap is received, it is possible to freeze the tempo with another key. Two other keys bump the speed of the frozen tempo up or down, like a kind of gestural accelerando/decelerando. Another added dimension is *velocity conducting*, which adds an offset to the velocity of the currently playing notes, determined by the velocity of the performer's tap.

A General Description of Jam Factory

Jam Factory consists of four players, each of which holds an input stream of pitches and durations, each with its own set of transition tables. In addition, each player has other output controls that employ algorithmic devices to provide other kinds of

Fig. 12. The states of the Jam Factory transition tables when computing pitches based on an input of C-sharp, D, E, D, F.



easily manipulated automatic variation. Controls for the players are located in windows in each corner of the screen. A long window in the center, called the *control strip*, handles starting and stopping the program's clock, and assigning the functions of MIDI IN and OUT for each player. The presets, which store the states of all the player controls, are housed in a window at the bottom of the screen. Some noninteractive screens are brought up from menu commands which configure memory use, set performance triggers, set player note limits, and create maps for scale and time distortion.

The transition table is a way of storing the probability that a certain action happened given some number of previous actions. The subject of transition tables and Markov chains has been well covered (Olson and Belar 1961; Pinkerton 1956). [See also Hiller 1970; Hiller and Isaacson 1959; Moorer 1972; Xenakis 1971—Ed.] Here I will review the basics and point out some techniques that assist in the real-time implementation used in Jam Factory.

The *order* of a table refers to the number of events in a "previous action" that the next action sitting in the table happened after. Figure 12 shows the entries in transition tables of orders 1–4 for an example input sequence of C-sharp, D, E, D, F.

An essential part of the Jam Factory algorithm is the probabilistic decision made on every note as to what transition table to use. This is possible because every time a note is recorded, entries are made in all the active tables; this creates something like multiple "views" of the same object.

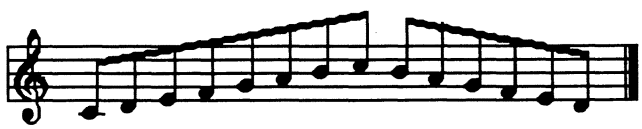
On playback, a buffer of the last four notes the algorithm has produced is kept around, so that the information necessary to "research" the previous states for any order transition table is available. Because of this, any of the tables can be used to generate the next note at any time. The probabilistic decision is based on a bar graph which can be manipulated directly by the user while the algorithm operates, allowing a fine tuning of its performance.

What is the effect of alternating between the use of one table and another? One way to illustrate it is with a major scale that ascends and then descends (Fig. 13a). When the bar graph is set at 100 percent Order 1 (Fig. 13b), the output exhibits a wandering based on the "local phenomena"—transitions of either up or down one note. As you begin to introduce Order 2 transitions (Fig. 13c) the output begins to take on more of the ascending then descending nature of the original input. Since an Order 2 transition table completely captures the information necessary to recreate the scale, no "mistakes" are made if the bar graph is set at 100 percent Order 2 (Fig. 13d).

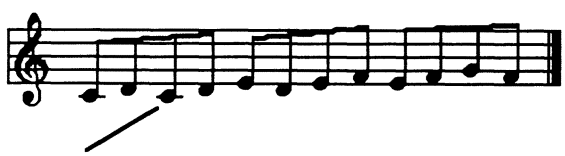
With more complex source material, it is not as

Fig. 13. The effects of transition-table processing. (a) Source material: an ascending and descending C major scale. (b) Sample output based on input of (a) for completely Order 1 transition table decisions. (c) 80 percent Order 2, 20 percent Order 1 decisions. Note that the output of (c) more closely resembles the input over a longer period of time than does (b). (d) 100 percent Order 2 decisions reproduce the ascending and descending scale exactly.

(a) Input source material

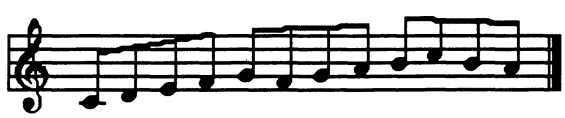


(b) First-order transition table output

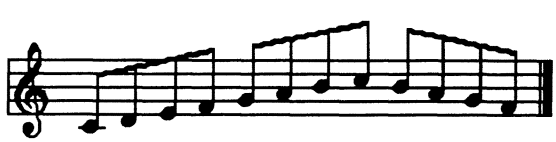


Starting over

(c) Primarily second-order transition table output



(d) All second-order output



easy to describe the effect of different settings of the bar graph, but it is not difficult with the feedback provided to try different configurations until one that produces reasonable results is found. 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.

The transition table scheme is duplicated for durations (the time between successive notes). Since there are an infinite number of possible durations, some kind of data reduction scheme has to be employed. Jam Factory has a quantization algorithm that forces notes into chords (pitch events in the transition tables can be polyphonic) and creates durations of at least 1 "unit." The unit of quantization is set as a note value (i.e., eighth notes). The more precise the quantization value, the "longer" each duration becomes, as illustrated in Fig. 14. Since the durations are expressed in units that multiply the time base, it is necessary to adjust the time base to equal the quantization value if the resulting music is to be played at the same speed as the original.

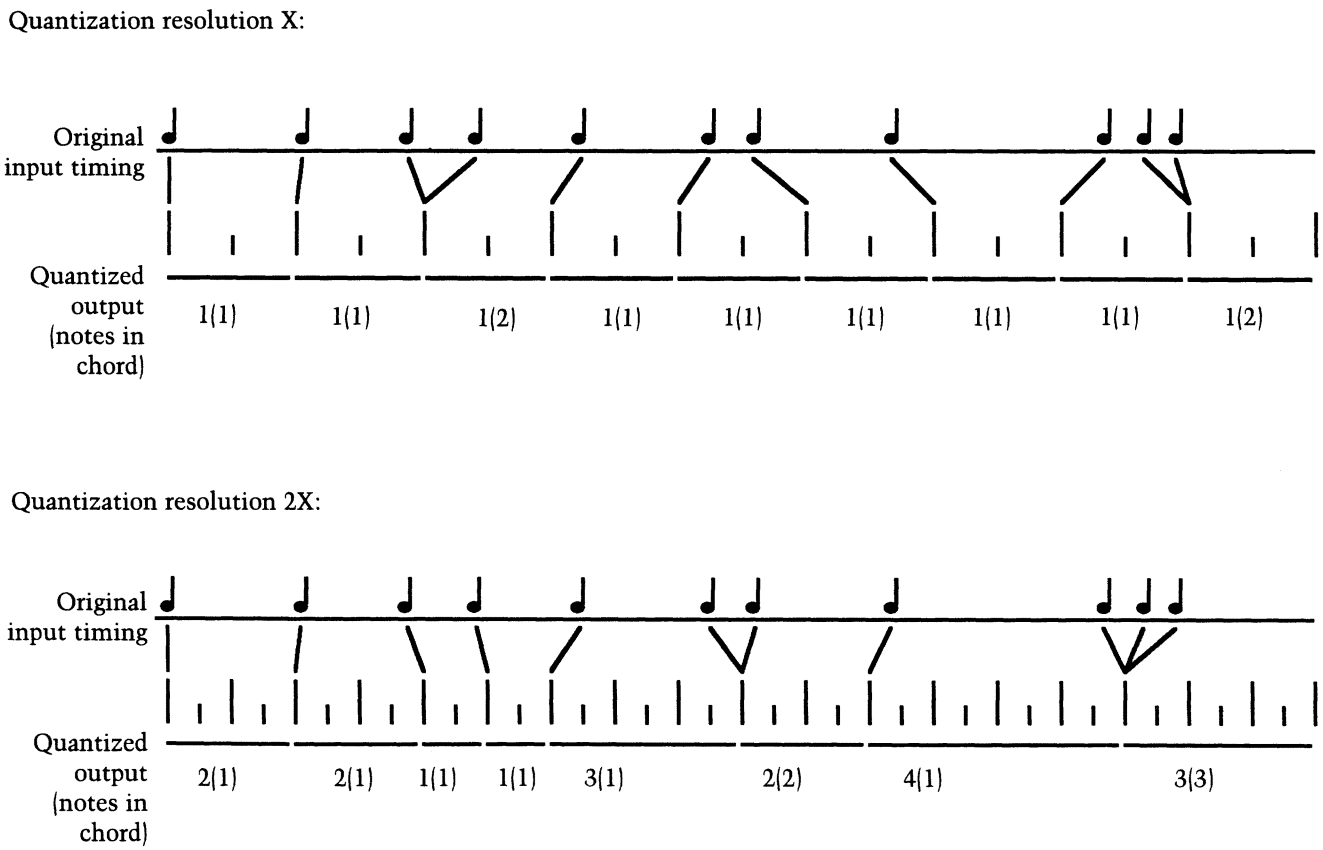
One particular enhancement of the duration scheme is worth mentioning. It combines information about a duration with the position of a duration within a cycle. As an example, a duration of 3 followed by a duration of 2 might turn into a duration of 3 on count 1 of the cycle, followed by a duration of 2 on count 4 of the cycle. This extra "position" constraint has the effect of insuring that notes of a certain duration always begin in a place in the cycle where they happened in the original input signal. A parameter in each player's window controls the length of this cycle.

Jam Factory provides objects for enhancing the output of the transition tables. The goal of the enhancements is to process the raw material and give it the air of an individual performance (although there is no provision for having the players actually listen to each other). The controls fall into these categories: the silences algorithm, velocity/articulation, micro timing, and live performance processing.

The silences algorithm is basically the same thing as note density in M, but it is represented backwards (the percentage is of silence, not playing), and includes a couple of useful switches that add variety to a performance. Skip, when enabled, causes the next note in the sequence from the transition tables to be "thought of" when it isn't played (producing the effect of skipping a note in the melody). Sustain, when enabled, sustains any notes that might be playing through a random silence, giving the impression that the notes have a longer duration, rather than there just not being a note there. Combinations of having these two effects on and off produce a variety of "playing styles."

Jam Factory's control over velocity and articulation for each player is very similar to the scheme used in M. There is a repeating pattern of levels, analogous to a cyclic distribution with the excep-

Fig. 14. Two possible quantizations of a sequence of note attacks showing the effect of increasing the value of each event's duration when increasing the quantization resolution.



tion that the levels are always deterministic. The absolute levels of velocity and articulation (percentage of time between successive notes before the note is stopped) are set using range bars. If the repeating pattern is not applied to one of these parameters, the values vary randomly within a range. If the pattern is applied, the values are extracted from within the range bar in the same way that M's intensity range interacts with the accent cyclic distribution.

Swing and time-distortion manipulations are efforts to move away from the feeling of a mechanized performance. Since they can be set differently for each player, the collective effect from all four players can be that of an ensemble, rather than that of a bunch of notes being triggered from a single clock. Swing is simply a distortion of the time given to two notes to emulate the long-short feeling popularly known as swing. It is expressed in terms of the percentage of the total time for the two notes taken up by the first note; thus, no swing would be 50 percent. In Jam Factory, unlike the implementation of swing (or "shuffle") on many drum machines, the value can range from 10–90 percent (values below 50 percent could be thought of as "inverse swing").

Time distortion is a more general-purpose method of manipulating micro timing. It is an implementation of the time maps proposed by David Jaffe as a method for producing the perception of ensemble performance (Jaffe 1985). The user creates time maps using a graphic editor, and sets a length of time over which the map will apply (maps repeat, like almost everything else). Maps that are the length of two time-base units in a player are equivalent to swing, while maps covering longer periods of time can be characterized as rubato. Since a player can have both some swing and an active time map, you can have a feeling of rubato on top of a feeling of uneven eighth notes (Fig. 15).

Time distortion is implemented by creating a

Fig. 15. Computation of timing information in Jam Factory.

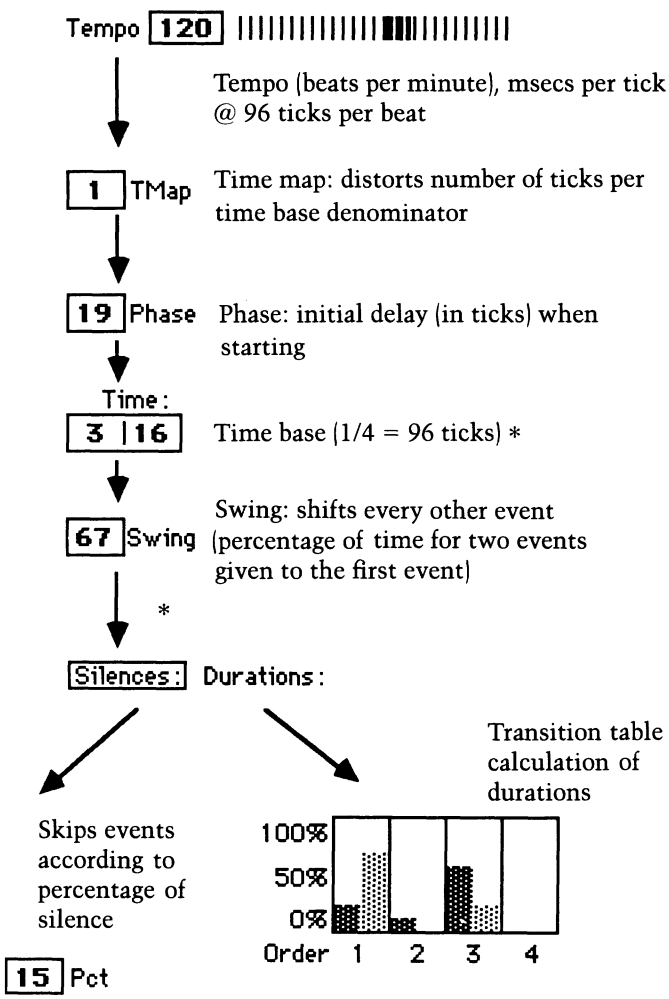


table that has as many entries as there are ticks (recall 1 tick = 1/96th of a quarter note) in the map. Every time the processing for a tick is executed, the table entry is read saying how many ticks to "simulate." If the map is slowing down real time versus clock time, there will be table entries of 0, causing the player to do nothing at that moment in time. When the map is speeding up real time, there will be entries of 2 (or perhaps more), in which case the player does twice as much as it normally would. Table entries of 1 do not speed up or slow down the player.

For live performance processing, the program does a certain amount of differentiation of MIDI input to make it possible to use multiple keyboards and/or performers. Each player has a column in the control strip labeled "In Ch" which can either be set to "All" or to a specific channel. MIDI input can be used in four ways:

- As source material for one or more players (i.e., "recording")
- To perform keyboard transposition (expressed an offset from middle C)
- As Jam Factory's version of the input control system
- To echo the input to the player's MIDI output channel

The assignment matrix in the control strip enables these tasks for each player's MIDI IN channel. The first three uses of MIDI input are mutually exclusive.

Control presets reconfigure the assignment of MIDI input. A control preset has a trigger (which can be any arbitrary MIDI event), which, when detected, changes the assignment matrix to a previously stored state. Thus you can go from recording to transposing without having to touch the Macintosh keyboard or mouse.

Keyboard transposition has an added wrinkle known as scale distortion. Using the scale distortion configuration window, you create remappings of the chromatic scale. For example, if the program were to play an F-sharp, you could force it to play an F. There are eight scale maps, which can be assigned to a key in the chromatic scale. When that key is played as a player is being keyboard-transposed, the map is activated. The result is that the tonality of the music changes but not the apparent melodic "motion." Yavelow (1987) observes: ". . . this process permits, for example, the transformation of highly chromatic music into diatonic music, quick conversion of music from minor to major scales and back, or styles which are scale/mode defined to be represented." A scale distortion map is implemented as a table of positive or negative offsets for each MIDI note number. If a map converted each F-sharp to F, the entry would be −1 for all MIDI notes that were F-sharp.

MIDI Files

Both M and Jam Factory implement the MIDI File standard. MIDI Files are a way of storing MIDI sequences in a way that is publicly documented and easily read and written, so that programs that deal with sequence data can exchange it. Usually, programs store data in formats besides the MIDI File format to preserve other kinds of information (the currently implemented standard has no notion of hierarchical sequence structure, for example). M and Jam Factory both have their own document format, which stores information such as the values and contents of M's variables and the mapping of MIDI events to entries in Jam Factory's transition tables. But importing and exporting sequence information is also supported, so that the programs can pass musical material to each other or to other programs that currently support the format, among them Opcode's Sequencer 2.5 and Southworth's MIDI Paint.

The programs contain a "movie" mode wherein every MIDI event that occurs in a performance is captured, along with the time it occurred. The events can then be saved as a MIDI File or, in M, regurgitated back into a pattern. The importation of sequence data is used as source material for the players in Jam Factory, and as real-time patterns in M. Since M allows one kind of pattern to be converted to another, data from a MIDI file can be treated with the same amount of flexibility as any other kind.

Dave Oppenheim, the author of Sequencer 2.5, developed the MIDI File format primarily because he realized that he wouldn't be able to implement every feature into his program that his customers wanted, and thought that it would be helpful if users who were also programmers could make their own use of the file format. The ability of M and Jam Factory to read and write MIDI Files has been largely responsible for their popularity with people who saw them as ways to make movies of what they often call "texture" to be used as tracks in their sequences. There is also the possibility of notating material created with the programs using Electronic Arts' Deluxe Music Construction Set in conjunction with the Opcode Sequencer.

The opportunities created by the MIDI File specification are applicable within the modular concept of MIDI in the largest sense. Prospective developers of music software are now free to work on a specific problem, knowing that there are other software packages which can take care of tasks they may not have the interest or time to worry about. It is, in other words, no longer necessary to write programs that purport to be "general purpose." We may get to the point where there are hundreds of little, specialized programs sharing information using MIDI Files; but this seems to me to be entirely healthy, since there are as many ways of using a computer to produce music as there are computer users.

Reflections

An audience listening to a "finished" piece of music, whether by purchasing recordings or attending performances, can be said to have "consumed" a product. Traditional music software programs, such as sequencers, are tools that make products. M and Jam Factory can be used to create fixed pieces of music, but comments from users (of the type "I'm having a great time with your program" more often than "I'm making great music with your program") indicate that these programs are an intertwined mixture of process and end-results. The programs provide enjoyment because their processes provide discovery and the pleasure of performance, and this enjoyment is enhanced if the user is satisfied with the music that is being produced.

In publishing M and Jam Factory, Intelligent Music's intention was to create, for the professional musician, enjoyable but powerful composing and performing tools which would enhance creativity. The value (and enjoyment) in these programs derives from the programs' participation in the composing process, which is to say that the programs share in making "decisions," at least in the sense of presenting to the composer or performer potentially usable ideas. But such a situation poses some ethical questions. If tools allow a composer or performer to produce music in a matter of minutes that might have taken weeks to do beforehand, should the composer or performer be rewarded for

this activity at the same level? Is this activity still "composing" or "performing" and is the result something that a composer or performer can "own," as a product? What credit should go to the program designer? Laurie Speigel asks that she and her program Music Mouse be credited in performances (Speigel 1986). To the authors of M, the issue seemed reasonably clear: their intention was to produce a general-purpose program, usable by any composer in any style, and their reward was to be the wide variety of music produced by users of the program.

The company had another intention, however, which was to test the applicability of the approach to amateur music-making. Just as the radio and record player allowed more people to enjoy musical performances, the interactive composition environment combined with an inexpensive synthesizer may allow a wider audience to experience the act of composing or improvising, in which case the "power" in the programs might make up for deficiency in performance skill. The potential benefits in this approach must yet be verified.

But then we are still learning. In making a program into a commercial product, you further its development tremendously through your interaction with a large number of users. However, the time involved in perfecting a new idea in the form of commercial software leaves you with years' worth of ideas and user suggestions to implement.

Acknowledgments

I would like to acknowledge Joel Chadabe for his help in the organization and editing of this article. The encouragement of Dr. Earl Schubert, my advisor in the Program in Hearing and Speech Sciences at Stanford University, was also very helpful. Many of the ideas presented here were formulated and refined in discussions with John Offenhartz and Antony Widoff, and I would also like to acknowledge the help provided by the reactions of those who worked with the programs as they developed, including Dave Oppenheim, Doug Wyatt, Jeffrey Rona, David Bluefield, Christopher Yavelow, and Jim Burgess.

References

Chadabe, J. 1984. "Interactive Composing: An Overview." *Computer Music Journal* 8(1):22–27.

Chadabe, J., and D. Zicarelli. 1986. *Jam Factory User's Manual.* Albany, New York: Intelligent Computer Music Systems, Inc.

Chadabe, J., and D. Zicarelli. 1987. *M User's Manual.* Albany, New York: Intelligent Computer Music Systems, Inc.

Hiller, L. 1970. "Music Composed with Computers—A Historical Survey." In H. Lincoln, ed. 1970. *The Computer and Music.* Ithaca: Cornell University Press.

Hiller, L., and L. Isaacson. 1959. *Experimental Music.* New York: McGraw-Hill.

Jaffe, D. 1985. "Ensemble Timing In Computer Music." *Computer Music Journal* 9(4):38–48.

Moorer, J. A. 1972. "Music and Computer Composition." *Communications of the Association for Computer Machinery* 15(2):104–113.

Olson, H., and H. Belar. 1961. "Aid to Music Composition Employing a Random Probability System." *Journal of the Acoustical Society of America* 33(9):1163–1171.

Pinkerton, R. 1956. "Information Theory and Melody." *Scientific American* 194:77–86.

Speigel, L. 1986. *Music Mouse User's Manual.* Palo Alto, California: Opcode Systems.

Xenakis, I. 1971. *Formalized Music.* Bloomington: Indiana University Press.

Yavelow, C. 1987. "Personal Computers and Music." *Journal of the Audio Engineering Society* 35(3):161–188.