Behavior Research Methods, Instruments, & Computers
1988, 20 (2), 255-262

A rule-based expert system for music perception

JACQUELINE A. JONES, BENJAMIN O. MILLER, and DON L. SCARBOROUGH
Brooklyn College of the City University of New York, Brooklyn, New York

We describe an expert system written in Pascal to simulate part of Lerdahl and Jackendoff's (1983) theory of music perception. This program, a production system based on a Hearsay II architecture (Nii, 1986), illustrates a number of techniques, including program modularity, complex data structures, and simulated parallelism using operating system concepts.

People perceive structure in the music they hear. Consider the simple song "Row, Row, Row Your Boat." Asked to sing this melody, most people would, without thinking about it, stress certain syllables, breaking the melody up into sections corresponding to the following lines:

ROW row *row* your boat
GENT ly down the *stream*
MER ri ly mer ri ly *mer* ri ly mer ri ly
LIFE is but a *dream*,

in which italicized syllables are stressed, capitalized syllables more than lowercase. The stresses are not random or idiosyncratic. The song divides into rhythmically similar halves, each consisting of two lines, with two stressed notes per line, equally spaced in time. This pattern of stresses is the *metric structure*. The division into lines is the *grouping structure*, also an important part of music perception. Listeners also perceive the tonality of the piece—the key it is in. After listening to a little bit of "Row, Row," a listener knows what chord to expect to end the piece. These are the kinds of intuitions that Lerdahl and Jackendoff (1983) formalized in *A Generative Theory of Tonal Music* (GTTM). The theory does this by means of four stages of analysis, each of which is embodied in a set of rules. This theory is, to our knowledge, the most comprehensive formal theory of music perception published to date.

We are building a computer simulation of GTTM to test the theory. The basic framework of the simulation is in place and currently consists of about 10,000 lines of code. Turbo Pascal was chosen for initial development of our simulation because of the many strengths of the language (see Jones & Harrow, 1986; Kaplan, 1985) and because it is a language that we know well. There are, however, problems with using Pascal for a large project with several programmers.

The authors are listed alphabetically. All contributed equally to this work. This research was supported in part by a National Science Foundation Graduate Fellowship awarded to Miller, and by a PSC-CUNY Research Award to Jones. We thank Hollis Scarborough for help in preparing the figures. Requests for reprints should be sent to Jacqueline A. Jones, Department of Computer and Information Science, Brooklyn College, Brooklyn, NY 11210, or to Don Scarborough, Department of Psychology, Brooklyn College, Brooklyn, NY 11210.

First, standard Pascal (and dialects such as Version 3.01 of Turbo Pascal that we have been using) provides limited support for modular development of large programs. All subprocedures are nested within the main procedure, and there is no separate compilation of subprocedures. Thus, if a line of code is modified, the entire program must be recompiled. Although compilation is fast with Turbo Pascal, the size of our program makes this time-consuming.

More important, standard Pascal does not allow a subprogram to define static variables—that is, variables that are allocated permanent memory locations for the life of the program. Thus it is often necessary to use global variables—that is, variables declared in the main procedure—for this purpose. However, with global variables, modifying a piece of code anywhere in the program may affect other aspects of the program in unanticipated ways. Also, name conflicts arise when several programmers are defining global variables for different subprograms.

To minimize such problems, we adopted several strategies for program development. First, we borrowed a Modula 2 approach (Wirth, 1985) to implementing the system. Each module of the program is split into two separate source files: a definition file (with a DEF file extension) and an implementation file (with an IMP file extension). The DEF file contains all the global variables whose values must be retained between calls to that module; these global variables take the place of static variables. The IMP file contains the actual implementation code for the module. To minimize name conflicts, we adopted a convention of prefixing all true global type definitions and global variables with an underbar. The names of global variables that mimic static variables—that is, variables that exist only for the use of the module in which they are defined—are preceded by the initials of the name of the module in which they are defined.

A small main source file defines the overall program structure. This file contains global type definitions and variable declarations that must be shared by all parts of the program. The main file then specifies the DEF files that are to be "included" in this file in a special step during compilation. Collectively, then, the main file and the DEF files specify the entire global environment of the program. The main file next specifies the IMP files as an

additional set of "include" files to be brought into the main file during compilation. Finally, the main file contains a short main procedure that calls an initialization subroutine and then passes control to the scheduling routine described below.

A RULE-BASED EXPERT SYSTEM

Although we considered implementing GTTM as a neural network (e.g., Grossberg, 1980; Rumelhart & McClelland, 1986; Schneider, 1987), and hope to do so eventually, there are several problems with the connectionist approach that led us to reject this approach for now. First, simulation of connectionist networks requires powerful computational resources for anything but the smallest networks. Second, the high degree of interaction between system components in a neural network makes modular development difficult. Third, many properties of these systems are not well understood. Fourth, many aspects of GTTM are not well specified or articulated. To attempt to implement a complex and incompletely specified rule-based theory (GTTM) in terms of a complex, interactive, nonsymbolic neural network is to risk chaos. Thus, our first goal is to see whether the theory can be adequately implemented at the symbolic level at which it is cast. However, as noted below, some parts of the simulation are easily cast into a connectionist framework.

We chose to develop this system as a rule-based expert system because (1) GTTM is expressed in terms of rules; (2) the technology for rule-based systems is developing rapidly, and there are many available techniques to call upon; and (3) a rule-based approach supports implementation of a modular system.

Although GTTM is expressed as a set of rules, many of the rules cannot be translated directly into simulation rules because, in general, the rules are not statements about how a piece of music should be analyzed, but rather are constraints that such an analysis should satisfy. A big hurdle in implementing GTTM lies in discovering psychologically plausible procedures that can produce analyses satisfying such constraints. We want the simulation to be plausible as a real-time implementation of the psychological processes underlying music perception. To this end, we chose to implement the theory as a rule-based production system (Newell & Simon, 1972; Winston, 1984), using a blackboard type of architecture (Nii, 1986).

Blackboard systems have been popular in psychological modeling (e.g., McClelland & Rumelhart, 1981). A blackboard system has three basic components: (1) a global data structure called the *blackboard*; (2) a set of *knowledge sources* (KSs), each of which encapsulates knowledge about some aspect of the problem; and (3) a *control structure* that monitors and controls which KSs are active. Our current implementation of GTTM in terms of these components is described below.

ARCHITECTURE OF THE SYSTEM

The Blackboard

The blackboard stores all the input data about a problem as well as information accumulated as the analysis proceeds. A blackboard is generally divided into several levels, each representing a different type of information. In our implementation, the lowest level of the blackboard contains data about the individual notes in the music. Other levels of the blackboard represent information about the analysis of the piece, including levels for grouping information, metric information, and tonality information, and levels based on higher level combinations of this information. In our implementation, information within each of these levels is stored in linked lists. Pointers between levels of the blackboard provide appropriate linking and synchronization between levels.

Our blackboard data structure had to be suitable for representing musical input. This representation had to be compact, because musical notation is very dense, containing thousands of notes, each supplying many different kinds of information. For this project, we assumed that the basic elements of a listener's experience include such information as pitch, note duration, dynamic level, and timbre. Thus, it seemed reasonable to build the simulation assuming this level of knowledge as a starting point, an assumption also made by Lerdahl and Jackendoff (1983). Input to the simulation program is a symbolic representation of such information, which forms the lowest level of the blackboard and represents what the listener hears.

We needed to represent the time continuum of the music as well as the multiple events taking place at each time, with some notes continuing to sound as others begin and others end. We used a matrix in which each row represents a different instrument (or voice) and the sequence of notes it plays, and each column represents the notes played by all instruments at a specific time.

A simple array representation would not be practical, given the limited memory of most personal computers, because it would be a "sparse" matrix (Tenenbaum & Augenstein, 1986), with far more storage occupied by blank information than by actual information. Suppose we wanted to represent piano music. At any moment, up to 10 notes may be played (more, if we consider sustained notes and cases where a finger plays 2 notes), so we might use an array with 10 rows, representing 10 possible notes at any time, and with columns representing successive notes in time. However, usually fewer than 10 notes are actually played at any time. An example of this, taken from Schubert, is illustrated schematically in Figure 1a. The passage begins with an 8-note chord held for two time intervals, followed by a sequence of 14 single notes. This general pattern is repeated eight times in the music. If we used an array of 10 rows and 16 columns to store the music, then in each occurrence of this rhythmic pattern

we would have an array with 160 cells, but 130 would be empty.

To avoid this problem, we used a sparse matrix implemented as a linked list. A linked list is a dynamic data structure that contains only those pieces of information that are necessary at any given time. An element of a linked list, known as a *node*, is a record that can contain several different kinds of data at the same time. An array exists in storage from the moment it is declared, whether it is used or not, but nodes in a linked list are allocated only as needed. The nodes are linked by pointers. Pointers—graphically represented as arrows—point to places where data are stored. One node can point to the next node or to the previous node, allowing us to create a list.

Figure 1b shows the same Schubert example as a sparse matrix using linked lists. Here, the top line represents nodes that simply mark off successive events in the music. Each box that is linked to an event is itself a node that stores information about an actual note. In this represen-

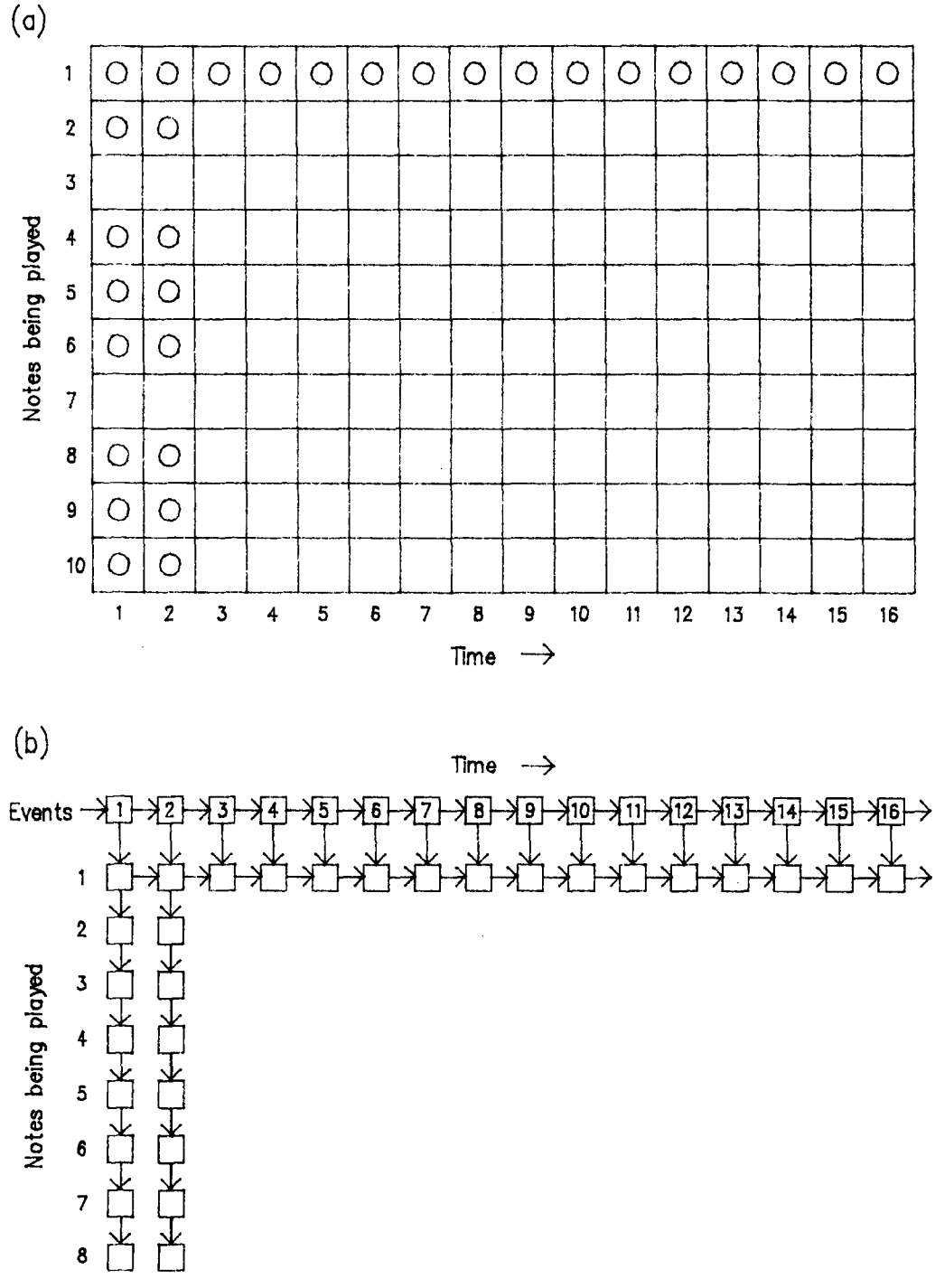

Figure 1. (a) Example of a "sparse" array with many empty cells. (b) A linked-list representation of (a).

tation, each occurrence of the Schubert pattern requires 46 nodes, compared with 160 entries in the array implementation. A further advantage of the linked-list representation is that it requires no prior commitment as to the number of notes that can be represented at any time or the number of notes per measure.

To simplify computation, the blackboard is a single global data structure that integrates all four facets of the simulation: the score, the metric structure, the grouping structure, and the tonality analysis. The "backbone" of the blackboard data structure is called the *notochord*. This consists of a doubly linked list of nodes corresponding to successive events in the score. Each of these nodes, called *event nodes*, contains only one piece of information about the piece: the offset (time) from the beginning of the previous event to the beginning of the current event. An event node also stores pointers connecting it to nodes containing actual note information, as well as to the grouping and metric representations.

Note that the event nodes themselves contain no score information. This is reserved for the *note nodes*, each of which is a record of all the information about a single note in a score. In addition to this information about each note, a note node contains a set of pointers that link it to other notes and to the notochord. The information in note 1 of "Row, Row" is shown in Figure 2 in simplified schematic form. (Note that there is no next voice because this is monophonic input. Assume that the soprano sings the melody.)

The Knowledge Sources

The GTTM rules represent sources of knowledge about how a piece should be analyzed. Where possible, we implemented each GTTM rule as a separate KS. A KS can be thought of as an "if-then" production rule; that is, a KS is a rule that specifies an action (or consequent) that applies whenever certain conditions (the antecedents or "if" parts) occur. An example of one of the simpler GTTM rules (Grouping Preference Rule 3d) is as follows:

IF $n_1n_2n_3n_4$ is a 4-note sequence,
and n_2 and n_3 differ in duration,
and n_1 and n_2 have equal durations,
and n_3 and n_4 have equal durations,
THEN there is evidence of a grouping boundary between n_2 and n_3.

In the GTTM simulation, such a rule is implemented as a KS module containing two parts: (1) a pattern matching procedure that looks at the blackboard for the specified antecedent pattern; and (2) a consequent procedure that makes appropriate changes to the blackboard.

To create a uniform interface between different KSs, we adopted a limited form of "object oriented programming" (Jacky & Kalet, 1987). Each KS is treated as an "object" that controls certain data structures, and that contains a set of methods for operating on these data structures. The KS operates on the data structure when it is sent a message specifying an operation to perform. We designed our KSs so that each responds to a limited set of messages (e.g., Initialize, Evaluate, Execute), thus facilitating a modular design that makes it fairly easy to add and replace KSs.

The Control Structure

The production rules (the KSs) operate under control of a scheduling procedure that monitors the activities of the KSs and the state of the blackboard. The scheduling mechanism reflects, in part, the fact that the simulation must be run on a serial von Neumann type of computer, where only a single KS can be active at any one time. However, the scheduling mechanism can also represent changes in attention and focus as the analysis of the piece proceeds. Furthermore, when appropriately implemented, the scheduler can simulate parallel concurrent activity of

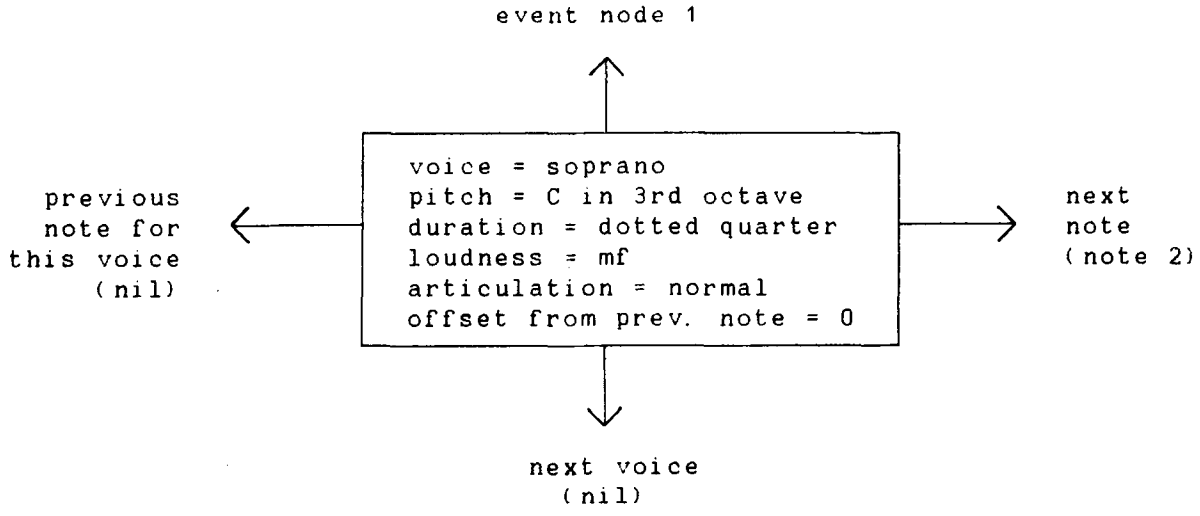

Figure 2. Schematic representation of the first note node for "Row, Row, Row Your Boat."

many KSs. The key to this aspect of our simulation lies in the window that specifies the portion of the blackboard that is "visible." A KS can examine only the portion of the blackboard within the window.

The window represents the capacity of short-term memory, which applies to music as well as to verbal material (Dowling & Harwood, 1986). In contrast, Lerdahl and Jackendoff (1983) offered analyses that assumed unlimited memory. Clearly, though, the perception of notes currently being heard is affected by how much a listener retains of the previous events. Therefore, our simulation is organized around the concept of a temporal window, so that in running the simulation we can vary parametrically the amount of prior input and analysis available at any point.

Initially, the program reads a file of information about a piece of music and builds the notochord data structure with the linked note lists to represent the piece. An initial portion of the notochord is then marked off by the window. The portion of the piece within the window represents what a listener first hears. This might be anything from 1 note to the entire piece, although currently we are working with a window size of 6 to 10 notes.

The scheduling procedure maintains a task table, which is an array of records, one record per KS. The array entry for a KS provides the scheduler with information about the status of the KS, such as whether it is currently runnable and its priority. A KS can run if the conditions it requires are currently met. The scheduler selects a KS on the basis of which KSs are runnable and the priority assigned to each. It then copies the task table entry for that KS to a global variable, where it is accessible to the KS, and then sends the KS a message specifying the action to be performed (e.g., Modify blackboard). The KS can modify entries in its task table entry to tell the scheduler about changes to its status and/or the actions it has performed. An outline of the system is shown in Figure 3, which also shows the Message Manager module that controls all output messages sent by the scheduler and the KSs.

This general framework for controlling execution lends itself to several scheduling algorithms. In round-robin scheduling, the scheduler simply goes down the list of entries in the task table, giving each runnable KS a chance to execute within the current window. Then the window is advanced (the new window may overlap the old and may exclude earlier events), and the scheduler calls each KS so the KS may evaluate its applicability to the new window. The scheduler then calls the runnable KSs again. A more complex algorithm schedules KSs on the basis of priority. In this method, the scheduler maintains a list of runnable KSs in order of priority and always calls the KS with the highest priority.

This scheduling procedure simulates parallel execution of the KSs in much the same way that multiuser operating systems such as UNIX produce apparent parallel execution of several tasks: The scheduler, which keeps track of all the tasks in the system, selects a task and allocates a small amount of time to it, and then selects another task for execution when the first one has used its allocated time,

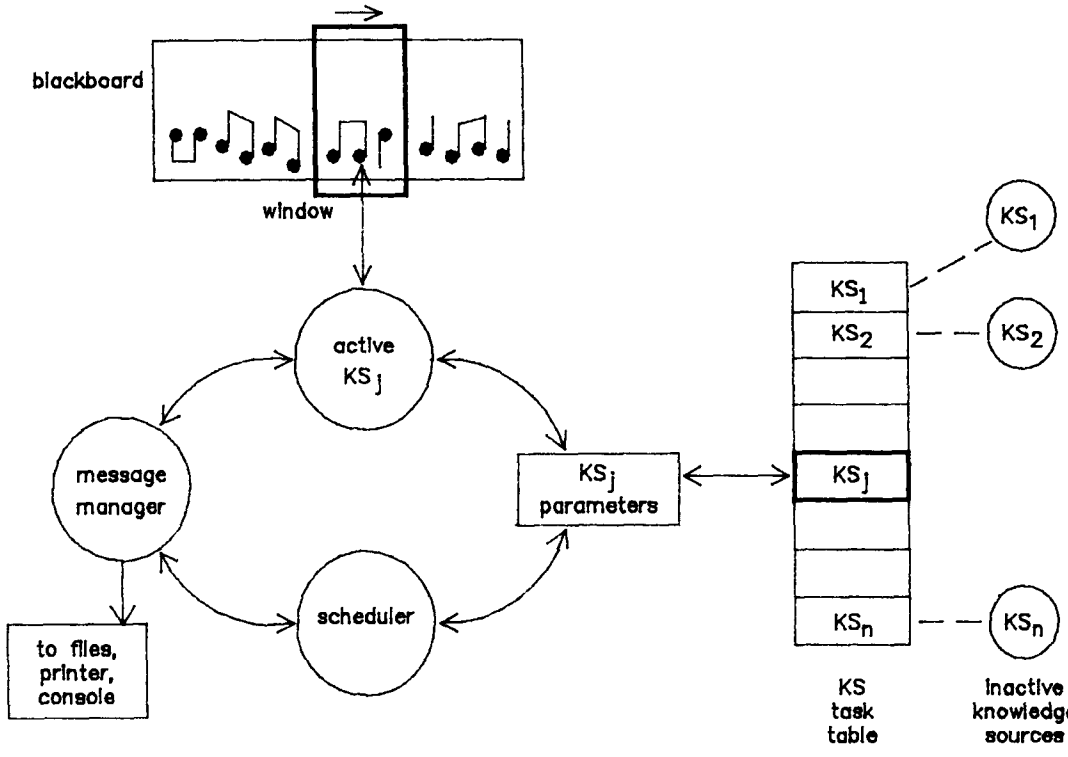

Figure 3. Control Structure for the simulation. KS = Knowledge structure.

finished, or blocked itself because of some other limitation (Deitel, 1983). In our simulation, the scheduler does not directly control the amount of time that a KS has for execution. Rather, the size of the window determines how much data a KS has to work with, and this in turn determines how long it will execute before running out of input data. With a small window, the effect is that all KSs appear to be executing concurrently and no KS can get very much ahead of or behind any other KS.

Scheduling permits more complex control if priorities are changed dynamically during analysis. Changing priorities is one way in which the system can incorporate a top-down component in the analysis: On the basis of earlier events in the piece, the system can learn which KSs are probably most appropriate, and this can be reflected in the priorities. As we develop the simulation, we will explore more complicated scheduling procedures. For example, the order in which several KSs are allowed to modify the blackboard can have important effects if the blackboard information is "nonmonotonic" (Rich, 1983), that is, if blackboard information does not simply accumulate, but can be removed or revised. In this context, we may explore other variations in blackboard architectures (e.g., Hayes-Roth, 1985).

Grouping Analysis

Grouping preference rules and subrules identify possible grouping boundaries in the score on the basis of such qualities as scale distance between notes, duration of time between notes, length of notes, and articulation of notes. Our prototype has KSs so far for two of the seven grouping preference rules, which comprise about nine subrules. Each subrule routine determines whether it finds a group boundary. If a subrule determines that it has found a boundary between two notes, a grouping node is created and added to the list of boundaries produced by that rule. Different grouping preference rules identify different boundaries, with varying amounts of evidence in their favor. This state of affairs must eventually give way to a single well-formed hierarchy of boundaries based on the strength of the evidence. This distillation will be carried out by higher level rules operating on the data structure(s) built by the grouping rules.

Each grouping KS creates a separate linked list of nodes running parallel to the notochord, one node for each candidate boundary it has found. Each grouping node is linked to the appropriate event in the notochord, as well as to the next grouping node created by the same KS, and to any other nodes produced by other grouping rules that have also identified a boundary at the same event. Linking all nodes supporting a boundary at a particular event makes it easy for a higher level rule to evaluate all the evidence for a particular boundary. This is illustrated in Figure 4, which shows the grouping analysis for "Row, Row." The boundary markers, attached to the note following the boundary, show results that replicate what we obtained by applying the rules by hand. The appropriate set of weights applied to the rules will select the conjunction of candidate boundaries between notes 11 and 12 (after "stream") as the most important. This boundary divides the piece into two half-songs. The boundaries between notes 5 and 6 (after "boat") and between notes 23 and 24 (after the last "merrily") will be selected as the second most important. As noted in the introduction, these boundaries also correspond to our intuitive feeling that the music divides into lines at these points.

Metric Analysis

The metric rules of GTTM cannot be translated directly into algorithms. That is, whereas the grouping rules themselves constitute parsing algorithms, the metric rules specify only what the metric hierarchy ought to look like when the analysis is finished. For this reason, we carry out the metric analysis in a single pass with a collection of routines that embody a single algorithm. The output of this algorithm conforms to the requirements of GTTM, but the algorithm itself is not taken directly from the GTTM rules. Instead, our approach is an adaptation of the grid theory of Povel (1984; Povel & Essens, 1985). Briefly, grid theory works by trying to fit different-sized

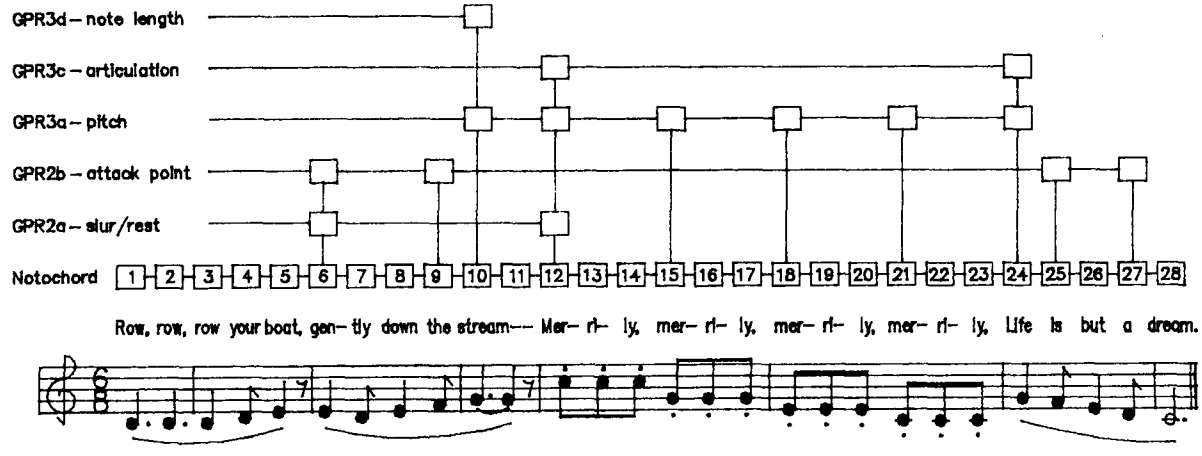

Figure 4. Grouping analysis of "Row, Row, Row Your Boat." GPR = grouping preference rule.

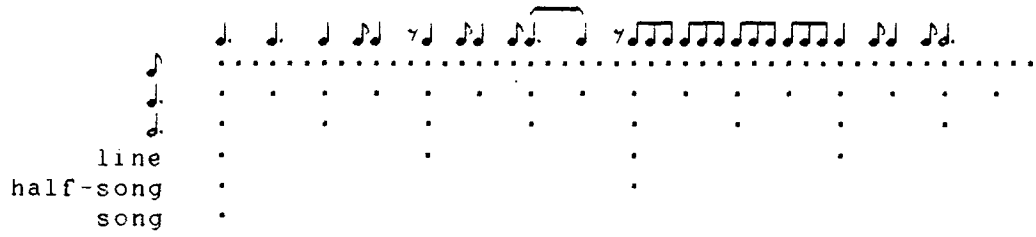

Figure 5. Metric analysis of "Row, Row, Row Your Boat." The number of dots indicates the relative rhythmic salience of the notes.

grids of equally spaced marks (grid ticks) to the musical events and then determining the degree of stress, or goodness of fit, between each of the grids and the events. We have modified the procedures of grid theory in several ways. For instance, only those grids suggested by the intervals in the music are considered. Each grid that fits corresponds to a metric level in the GTTM hierarchy. Our use of the grid stress is not to identify the one best grid, but rather to eliminate from the hierarchy those levels that least fit the music, that is, levels not heard by a listener.

The representation of metric structure is analogous to that outlined above for the score. Metric nodes contain information about the metric hierarchy at each beat, as well as pointers to other items and to the notochord. Here again, the linked-list structure saves space. If a metric hierarchy were represented by a two-dimensional array it would be quite sparse. By allocating a node for each beat rather than for each beat-by-level combination, the linked list of metric nodes wastes no space. The metric analysis of "Row, Row" is shown in Figure 5.

Tonality Analysis

Listeners can identify the key of a piece, and this can affect both meter and grouping. For example, strong beats in the metric structure generally fall on the important pitches of the key, as is clear in the "Row, Row" example, in which the accents generally coincide with the notes C, E, and G. On a piano keyboard, there are 12 black and white keys in each octave, representing the chromatic scale. In any one piece of music, some of these notes are more important than others, reflecting the key in which the piece is written. Thus, "Row, Row," which is in the key of C major, contains 25 notes, but only 5 (C, D, E, F, and G) of the possible 12 chromatic notes occur. C occurs eight times, E occurs five times, and G occurs five times. Thus, the notes C, E, and G account for 18 of the 25 notes, reflecting the fact that C, E, and G form the C major chord, which is the fundamental (tonic) chord for the key of C major.

Lerdahl and Jackendoff (1983) did not discuss how listeners extract the tonality of a piece, but the example just given suggests a simple algorithm: If we count the frequency of the notes of the chromatic scale, the most common notes will generally identify the key of the piece, as Simon (1968) has shown. Bharucha (1987) adapted this general idea to a network model, and we have followed a similar approach. In our implementation, each note of the chromatic scale is represented by a *pitch node*. These pitch nodes are connected to *chord nodes*; each chord node receives input from just three pitch nodes. Finally, sets of three chord nodes (representing the tonic, subdominant, and dominant chords of a particular key) are connected to *key nodes*. As the analysis of a piece progresses, pitch nodes are activated to the degree that their notes occur in the input. These pitch nodes in turn activate the chord nodes, which in turn activate key nodes. The activation level of each node is also affected by a decay parameter, so the activity of a node dies down if input to that node is not sustained, thus defining a different sort of temporal window. The most active key node at any point in the piece defines the perceived key at that point.

This tonality extraction mechanism is implemented as a single KS that is triggered by the appearance of new notes in the window. The tonality KS posts the current best hypotheses about the tonality on the blackboard by creating tonality nodes, with each new input event leading to the creation of a new tonality node when this KS is run.

DISCUSSION

Our research program addresses basic questions about music perception, but it also illustrates a number of programming techniques that should be of interest to psychologists working in many areas, especially in the study of speech perception, reading, and other types of temporal pattern perception. First, organizing a simulation in the manner of a multitasking operating system allows the simulation of parallel processing within the context of a highly modular program structure. This permits easy experimentation with a minimum of reprogramming. Second, the use of linked lists permits a data structure that efficiently stores information about temporal patterns in which the number of potential events far exceeds the number of actual events. In effect, linked lists take the sparseness out of a sparse matrix. Third, an input window is essential for simulating processes in which stimulus patterns exceed the span of the psychological present, because a program that has access to all the data is not a reasonable psychological model. Finally, the programming language must be equal to the task. We chose a powerful, widely known language that supports the complex data structures and sophisticated programming techniques that are essential in projects of this kind.

REFERENCES

BHARUCHA, J. J. (1987). MUSACT: A connectionist model of musical harmony. In *Program of the Ninth Annual Conference of the Cognitive Science Society* (pp. 508-517). Hillsdale, NJ: Erlbaum.

DEITEL, H. M. (1983). *An introduction to operating systems*. Reading, MA: Addison-Wesley.

DOWLING, W. J., & HARWOOD, D. L. (1986). *Music cognition*. Orlando, FL: Academic Press.

GROSSBERG, S. (1980). How does the brain build a cognitive code? *Psychological Review*, **87**, 1-51.

HAYES-ROTH, B. (1985). A blackboard architecture for control. *Artificial Intelligence*, **26**, 251-321.

JACKY, J., & KALET, I. (1987). An object-oriented programming discipline for standard Pascal. *Communications of the ACM*, **30**, 772-776.

JONES, J. A., & HARROW, K. (1986). *Problem solving using Turbo Pascal*. Englewood Cliffs, NJ: Prentice-Hall.

KAPLAN, H. (1985). Design decisions in a Pascal-based operant system. *Behavior Research Methods, Instruments, & Computers*, **17**, 307-318.

LERDAHL, F., & JACKENDOFF, R. (1983). *A generative theory of tonal music*. Cambridge, MA: MIT Press.

MCCLELLAND, J., & RUMELHART, D. (1981). An interactive model of context effects in letter perception: Part I. An account of basic findings. *Psychological Review*, **88**, 375-407.

NEWELL, A., & SIMON, H. A. (1972). *Human problem solving*. Englewood Cliffs, NJ: Prentice-Hall.

NII, H. (1986). Blackboard systems: The blackboard model of problem solving and the evolution of blackboard architectures. *Artificial Intelligence Magazine*, **7**(2), 38-53.

POVEL, D.-J. (1984). A theoretical framework for rhythm perception. *Psychological Research*, **45**, 315-337.

POVEL, D.-J., & ESSENS, P. (1985). Perception of temporal patterns. *Music Perception*, **2**, 411-440.

RICH, E. (1983). *Artificial intelligence*. New York: McGraw-Hill.

RUMELHART, D., & MCCLELLAND, J. (1986). *Parallel distributed processing: Explorations in the microstructure of cognition*. Cambridge, MA: MIT Press.

SCHNEIDER, W. (1987). Connectionism: Is it a paradigm shift for psychology? *Behavior Research Methods, Instruments, & Computers*, **19**, 73-83.

SIMON, H. A. (1968). Perception du pattern musical par AUDITEUR. *Sciences de l'Art*, **V-2**, 28-34.

TENENBAUM, A., & AUGENSTEIN, M. (1986). *Data structures using Pascal*. Englewood Cliffs, NJ: Prentice-Hall.

WINSTON, P. (1984). *Artificial intelligence* (2nd ed.). Reading, MA: Addison-Wesley.

WIRTH, N. (1985). *Programming in Modula-2* (3rd ed.). New York: Springer-Verlag.