KSEM: An essay in knowledge representation in music
Richard A. Ashley
Computer Music Studio
Northwestern University
Evanston, IL 60201
Tel. (312) 491-3685

ABSTRACT

KSEM, a rule- and fact-based program for representing and using musical knowledge, is described. An overview of the production system formalism for knowledge representation is given, followed by some observations on the applicability of this formalism to musical tasks. The organization and facilities of KSEM are described; a sample application written with the system is considered. Strengths and limitations of the formalism and of the program itself are examined, along with directions for future work.

1.0 INTRODUCTION

My research for the last several years has focused on cognition and on computer applications in musical tasks (see Ashley, 1982, in press). In the course of this work I have dealt with issues of the nature of musical knowledge and how it may be represented in computer programs. This has led to a desire on my part for a single set of programming tools for use in cognitive modelling and in "applied" knowledge representation (such as computer-assisted analysis). This paper describes KSEM (Knowledge System for Music), a system for representing and using musical knowledge based on the production system formalism. KSEM is designed to give a basic set of knowledge representation tools functioning either as an element of a computer music system or as a stand-alone program. The paper begins by reviewing the concepts of a production system architecture, and continues by considering some aspects of the nature of musical activities and their implications for knowledge-system design. The KSEM system is outlined, followed by the examination of an example programed with KSEM. Finally, an evaluation of the strengths and limitations of the approach in general and the system in specific is offered.

Initial development of KSEM was carried out while I was a member of the XSEM Sawin on Music and Technology held at Dartmouth College, 1984. I wish to thank XSEM, Jon Appleton, Gary Hagner, Gary Kangeli, Allen Winfield, and Terry Zippy for various kinds of help. I, of course, assume all responsibility for error.

1/ psi: A production system (abbreviated PS) is one way of representing knowledge in a computer program. A PS has three major components. The first is a set of rules, conditional statements of action to be taken should some state of affairs arise; a working memory or control database, on which the rules operate and through which they communicate with one another; and an interpreter or overall control structure which coordinates the actions of the system. Within this general format a variety of approaches have been used. The following gives more detail on the basic units of a PS.

A rule, also known as a production rule or a fact, is of the form:

IF <condition(s)> THEN <rule(s)>. The basic action of such a module is to wait for some state of affairs to be attained and then to carry out one or more specified actions. In a basic or "pure" PS the conditions involve only recognizing and matching elements in the rules' conditions with the contents of working memory. Many systems, including KSEM, allow such conditions in addition to this fundamental one through the use of predicates in rule conditions. The most typical actions are to add to, remove from, or change the contents of working memory, or to read a message from or write a message to a user at a terminal. A "real-life" rule, expressed in English, might read as in Ex. 1.

IF you have a fever of higher than 100°F and you hurt all over
THN suspect that you have the flu and call a doctor
and rest in bed
and take aspirin
and keep your fluid intake high.

Ex. 1: Sample rule

Notice that in this rule there is more than one condition and more than one action; and that the form of the conditions (the left-hand side or LHS) is conjunctive. Disjunctions are expressed by rules with similar but not identical LHS or RHS elements, rather than being incorporated in...
single rules. A PS may use many such rules, each waiting for its LS to be satisfied in order for its RS to be applicable.

This is the main component of a production system in working memory (abbreviated here as WM). This is a collection of assertions about states of affairs at a given point in time. Working memory may be organized in a number of ways, from simple list structures to highly complex databases or networks. The LS elements of rules generally refer to elements contained in WM; the RSs of rules alter the contents of WM so that other rules may then be applicable. In the rule above, one WM element which would be modified by a LS condition could be represented in a list structure as

\[ \text{BOOSTTEMP}(10) \]

The RSs of the rule would, among its other actions, add to WM the assertion that:

\[ \text{PROBABILECONDITION}(R) \]

which would then be available for further use by the LSs of other rules. WM can thus be viewed as a communication channel between independent procedures, each of which is embodied in a rule. The channel changes contents through the actions of rules, as they add and remove elements from WM in a discrete set of steps or stages. This communication between independent procedures, carried out through a single channel, is one main characteristic of a PS architecture.

The third main element of a production system is the interpreter. The interpreter has a number of responsibilities: it looks for input to the system, finds applicable rules, and, in the case that more than one rule is applicable in a given situation, decides which rule(s) to use. This process is in a main control line called the recognize-set cycle, involving the stages shown in Fig. 2:

- **RECOGNIZE:** (find applicable rules)
- **EXECUTE:** (execute chosen rules)
- **UPDATE** (change rules)

\[ \text{Ex. 1: Recognize-set cycle} \]

The RECOGNIZE stage looks for rules whose LSs are satisfied; that is, whose LS conditions have been met. Rules which pass this stage of processing are put into the conflict set. The next stage, Conflict resolution, chooses between alternative courses of action and helps to provide for a system which is both responsive to the environment and stable in changing circumstances. Conflict resolution is one of the major overall control aspects of a PS's organization. Finally, the appropriate rule is "fixed" or executed by passing the actions indicated in its RS. This loop continues until no rules are applicable or until the task at hand has been worked through.

PSs have been used for a wide variety of tasks, from cognitive modeling (Anderson, 1976, 1983; Newell, 1973) to "pure" knowledge-engineering tasks (Meredith, 1985); others can be found in Hesk & Fleischman (1982) and in Wertsch & King (1973). Systemm (1976) proposed a theory of PSs as a general programming language, which has several "classic" AI programs as PSs. The OPS production system (language developed at Carnegie- Mellon has been described in Ford & McCarthy (1973); McDermott & Nutting (1975) is an excellent discussion of conflict resolution.

3.0 MUSIC AND KNOWLEDGE REPRESENTATION

In choosing a knowledge representation scheme for use in some domain it is necessary to choose tools and techniques which are well-suited for the structure and activities of the domain. There are a relatively large number of tools and techniques available, each of which has its own advantages.

Many of these tools have in some degree of computational generality, such as Turing machine equivalence; it seems that researchers pick their tools according to considerations such as ease of use, a desire for richness of primitives or (or elegance, and in the nature of the task at hand. An entire issue of SIGART Newsletter was published a number of years ago (Strachey and Smith, 1980) which pointed up the diversity of opinion in the field; Karn & Fleischman (1981) also has summaries of the main techniques. A brief consideration of musical activities will help to show some of the reasons why production systems are a plausible means for representing musical knowledge.

There are a wide variety of musical tasks, such as composition, listening, theory, notation, and learning. All of these deal with time-variant processes and changing contexts or states of knowledge. The production system provides, in the recognize-set cycle and the changing states of WM, a means of specifying the environment.

Each time through the cycle the system is (at least theoretically) capable of responding to new phenomena, provided there are rules in the system which are assigned to the new circumstances.

Another aspect of musical activities is their goal-orientation. Musician often undertake some musical task without intending to attain some goal: new knowledge, a new composition, a new piece which can be performed for others, or enjoyment from listening are all typical goals for those involved with music (some might be related to others). Goals may be represented in a PS as a number of actions. Some systems, such as MOCO (Shortliffe, 1976), are specifically goal-oriented through the use of a main control strategy called backward chaining in which the system attempts to find rules which must be satisfied to lead to a goal. Alternately, goals may be included explicitly in having the rules reference them explicitly; this approach is taken by KSS and by Anderson (1976, 1985).

A third attractive feature of production systems for knowledge representation is their ability to represent the essential modularity of knowledge in the system. Each rule is a specialized bit of knowledge, which acts independently of the overall
coordination between rules and control of the system is left to the interpreter. In a field such as music, where we know relatively little about the nature of musical activities, the possibility of adding new information to the system incrementally as new discoveries are made is attractive.

One further aspect of music is of fundamental importance to KSM’s origin: the use of at least three main forms of representation and discourse in musical activities. Everyday musical life makes use of language, music notation, and sound itself to deal with different aspects of music. Even the simplest musical task may, on inspection, involve some rather opaque processes of translation between these. This being the case, KSM adds other special representations of data and knowledge to the basic production system structure to help address musical issues. This principle of using different forms of knowledge representation in a single system has been proposed for knowledge-based systems in general; see Noonan, 1983 for opinion on Chuc. A central working area like a P’s WM is one solution to the need to combine and coordinate various representations in, and aspects of, the system.

All Overview of KSM

KSM is a variation on the basic P’s structure. The intent of KSM is not to supply all possible elements of a knowledge representation system, but to give as small and manageable a system as would be useful. To this end, KSM uses a body of rules, a WM, and an interpreter similar to the basic one outlined above, but includes other components as well. In particular, it includes a database of facts, a means for representing music, and a set of utilities for manipulating the elements of the system. The diagram below illustrates the overall organization of KSM.

```
<table>
<thead>
<tr>
<th>Interpreter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fact base</td>
</tr>
<tr>
<td>Rule base</td>
</tr>
<tr>
<td>Working memory</td>
</tr>
<tr>
<td>Score storage</td>
</tr>
</tbody>
</table>
```

Fig. 1: KSM architecture

The current version of KSM is 1.0, superseding two previous versions, and is written in LISP running under MS-DOS. The decision to work on microcomputers rather than in a minicomputer or mainframe environment was dictated by a desire for portability and a relative lack of features available on Lisp implementations for the target computers at my previous university. KSM will now be rewritten in a standard LISP dialect (Franz LISP) for use on the Pyramid/Unix system in the Computer Music Studio at Northwestern. The following outlines the system as of July, 1984.

4a Rules, working memory, and interpreter

Conceptually, KSM rules are expressed a bit like record structures, with a field or slot for each element. The elements of a rule are:

1. A name (atom)
2. A goal (list)
3. A LHS (list)
4. A RHS (list)

Rules in KSM have the following format:

```
<rulename>
IF (goal) is active in WM
and (condition(s)) are in WM
THEN (case specified (action(s)).
```

A stylized English representation of a KSM rule is shown below.

```
RULE1
IF the current goal is to determine the overall style of the music being heard
and no discernable beat is found in the music
and the music is in highly chromatic
THEN add that the music is probably "modern"
and set as a subgoal to try to identify
the composer
and tell the user what the new goal is.
```

Ex. 3: Hypothetical KSM rule

The rules are represented as properties on the property list of the rule’s name rather than as an ordinary list structure. Thus, a given element of a rule is accessed by

```
(GET (rulename) (n))
```

The effect of this is to parcellize the nature of the elements in rules, allowing for access through aonomic labels, rather than by a position in a list structure.

KSM has a two-stage evaluation of a rule’s LHS: First the goal is checked against the currently active goal as a quick pre-sorting procedure; then the remainder of the LHS conditions are evaluated if needed. The goals which the system is addressing are found in a special area of WM acting as a stack, where current and pending goals (in goal-stack chain) are stored.

Rules in KSM are stored in tiles, allowing for two kinds of user partitioning of rule-based knowledge: implicit and explicit. The user may group rules intentionally by placing them in separate files according to their intended use. In this fashion a number of types of storage and processing overhead can be alleviated. The rules in a given tile can be loaded into the system as needed, either replacing or augmenting those already included. Through the use of the "goal" construct in the LHS of a KSM rule, rules may be partitioned into groups according to their

385

ICMC '85 Proceedings
area of application, and also partitioned hierarchically if the system developer has sufficient knowledge of goals and subgoals in the task under consideration. Rules are also partitioned implicitly and dynamically through the Recognize cycle, as those which are useful to the task at hand are found in the working process and chosen in conflict resolution.

Working memory in KSM is composed of a collection of elements, each of which is a list. Elements are generally added to or removed from WM through the actions of rules. Each element stands alone and is linked with no other. Unlike a "pure" production system, however, KSM does not have a truly undifferentiated WM. Rather, in KSM there are a number of main divisions, including:

- WM proper: a collection of list elements
- MYSQLSTATE: a stack for storing goals and subgoals
- INPUT: a flag indicating the presence of user input
- SCORES: a data structure for storing music.

This division of WM is not essential but helps a user to inspect and manipulate its contents. Elements in WM have two points of origin, with some coming directly from an ADD action in a rule's RHS and some from retrieval of facts from EM's fact base. The nature of the fact base will be described after KSM's interpreter is considered.

KSM uses a variation on the basic recognize-cycle cycle in its interpreter, in that it adds a couple of steps. These are checking for user input during the cycle and performing some bookkeeping after a rule has been ACTed upon. The first action taken by the interpreter in a rule is to check the INPUT flag's value: if there is user input, which needs attention, this is handled before proceeding with evaluation of LHS in the rule base. The user may, for instance, have requested some change in the system parameters; such requests are processed immediately, and control is then returned to the main rule-based activity.

Efficiency is a major concern in the design and implementation of production systems. One of the main focuses of this in the extremely large amount of processing time needed for pattern matching in the LHS of rules in order to determine the conflict set. KSM addresses this concern in part through the use of the "goal" construct in a LHS as a quick pre-check on a rule's applicability before the rest of the RHS has to be considered. Matching in the rest of the RHS can take place in two ways:

- "with the occurrence" of a "wild card" variable;
- with the non-NULL test of a predicate.

Kot includes a variety of predicates, including the typical arithmetic ones, tests for equality, and others. One such predicate which is not a built-in LISP function is SIMILARP. This is a function which compares two items and, depending on their type and a number of features, decides if a match is "good enough" for

\[
\text{(SIMILARP (item1) (item2))}
\]

to return a non-NULL value. All rules in which all LHS conditions have been satisfied are collected and placed in the conflict set. Conflict resolution in KSM is a multi-stage process, using the following steps if a single rule for execution in one interpreter cycle:

1. Refraction. Rules previously matched are prevented from doing so with the same WM elements. Without this step, endless loops are possible, as for example when no new rules match and old rules "try again."
2. Recog-R. The rule reacting to the most recently added WM element is chosen.
3. Specificity. The rule meeting the most stringent requirements is chosen.
4. Arbitrarity selection. If no single rule has yet emerged, one is chosen randomly.

These rules apply one after the other until only one rule is left, which then executes. With rulesets the size of the size now is not (to no more than a few dozen) this combination of conflict resolution rules has been satisfactory.

The ACT stage of the interpreter is straightforward. Some of the primitive actions available in KSM are:

- ADD to WM
- REMOVE from WM
- POSITION to the goalstack
- POP GOAL from the goalstack
- SEND a message to the user
- RECEIVE input from the user
- RETAIN the fact base
- EXIT from the task at hand
- Generate a new fact base
- KEETEN VM items to avoid "swamping" (q.v.)

The arguments of these actions vary as needed; for example, SEND takes a string or list as its single argument, whereas KEET takes no arguments.

After the appropriate KSM actions have been taken, KSM performs some bookkeeping. The list of previously applied rules (and in the "refraction" stage conflict resolution is updated as needed, an old and unused VM elements are "wiped" (exterminated) from WM to reduce overhead and unnecessary pattern matching time in LHS evaluation. The cycle then repeats as needed.

4.7 Fact base, music base, user interface, and utilities

A production system focuses on knowledge as represented in procedural as opposed to declarative form. In a PS, rules embody the system's domain knowledge, the interpreter contains the control knowledge, and the working memory serves as an adjunct to these. Thus, knowledge is primarily about when and how to do things. KSM deviates from a "pure" production system in including a major declarative knowledge
component, called the fact base. A fact is an assertion of proposition, such as

(BOOTSPEL PEP)

Many such assertions are, of course, found in Rete of rules and may be added to KB in the ACT protocol. The difference in ESM is that facts are added conditionally in groups, called contexts. A context consists of a collection of facts in which are added with some together upon a RETE New Act, being executed. Context in ESM are arranged in a hierarchy like fashion, but in constructing sequences between levels and columns. The hierarchy created from contexts in the ESM fae base may be thought of as three-dimensional, with connections crossing between planes.

A context in ESM is composed of a number of elements:

1. A name or label;
2. A list of facts;
3. A list of links to other contexts, each link showing the name of the connected context, and also the strength of the connection.

Links are at this time always explicitly declined by a user. The link strength is used by ESM as a heuristic for using facts: less strongly-said facts are more quickly swept from KB unless used or reeaved, eliminating excess space and pattern-matching requirement.

The fact base in ESM is not, strictly speaking, necessary. Facts can be handled in a GP by using rules with wild-card LHSs and many sides in the RHA (this method was used in various 0.0 and 0.1 of ESM). However, the inclusion of a declarative component seems natural and useful in a number of ways. My own opinion in using these iterations of ESM is that having this distinction available is much more user-transparent than, for example, dealing with rules with null LHSs. In addition, rules with null LHSs may take up space in a rule base and processing time for LHS evaluation and conflict resolution unless some kind of pre-evaluation filtering takes place. In contrast, the awareness adding and sweeping of facts in ESM is reasonably efficient. There is also some evidence from cognitive psychology that seems to support the viewpoint that declarative and procedural knowledge are different in their acquisition, internal representation, and use (see Anderson, 1983 for discussion on this subject). Since my initial interest in production systems was for cognitive modeling and ESM needs to continue to support this kind of endeavor, the fact base is a reasonable solution.

The fact base is dictated by the general principle that knowledge should be represented in as felicitous a manner as possible, even at the cost of some parsimony.

The part of ESM currently undergoing the greatest change is that dealing with internal representation of motifs. Version 0.0 and 1.1 of ESM had very simple formats for storing music, consisting of lists of notes with pitch, octave, and duration indicated, as well as rests, bar lines, clefs, and key signatures. For these formats was adapted to facilitate a normal level of communication when dealing with traditional musical concepts and constructs. As the purpose of ESM is not to provide a music editing system but a knowledge representation system these have been sufficient; now however, their limits have been reached. As of this writing the system is being redesigned to conform with the event-list format in use at the Center Music Studio of Northeastern University, which provides a flexible and highly general means of editing and representing music. Use of the event-list format will also allow ESM to function as a unit in a full array of composer music software, allowing the construction of knowledge-based composition, synthesis, musical, and analytical programs.

The overall goal of music representation in ESM is having a way of storing, accessing, and working upon music which is hardware-independent (so that the system may be portable) and which facilitates discourse about music. To this end, rules and facts in ESM contain information about music notation and structure which allow translation from internal format to English-like user descriptions.

Designing a user interface for ESM has been a matter of balancing development costs, ease of use, and ease of learning. In its present form ESM makes use of hierarchically-structured menus to guide the user through the program. The main menu gives the user a number of options:

1. Set system parameters;
2. Rule editor;
3. Fact editor;
4. Music editor;
5. Variables editor;
6. Execute a program run;
7. Quit.

Each entry on this menu leads to more major utility, such as the rule editor, fact editor, or music editor. The options on the rule editor menu are given as follows:

1. See a rule;
2. Edit LHS;
3. Edit RHS;
4. Make a rule;
5. Set a goal;
6. Find rule(s) with element X;
7. Quit to main menu.

Some utilities in ESM are available only when the user is in the rule editor, fact editing, or the rule and fact editors; others (e.g. the entry in the menu that allows the user to set a goal) may be used from the main menu. These run-time utilities include:

1. A "window" on KB for viewing its contents;
2. A means of inspecting the conflict set;
3. A check on how slow a rule is to be being satisfied in the current interpreter cycle;
4. A means of inspecting the state of system variables, such as scoping which have been accessed;
5. A "why" facility which allows the user to
see the chaining of rules and goals in the system. These are useful in making the workings of the system more transparent to the user, especially when unexpected results occur. This is of prime importance in a KS, where the flow of processing and data in the program are more opaque than in any other styles of programming.

5.2 A SIMPLE KS APPLICATION

The foregoing has described KS in general and given some particulars about the goals and organization of KSM. This section gives a concrete example of a small application written with KSM. The following are illustrated:

- How goal-oriented behavior is attained in KSM;
- How facts and rules are used in KSM's operation;
- How different levels of discourse may be represented in KSM.

(In this example the working elements of the system are shown in stylized English. Readers wishing to see actual examples of KSM rules and VM elements may write to me.)

The example is given a melody (from that favorite work of analysts, Mozart K.331, first movement) to analyze. The excerpt under consideration is shown below.

Ex. 4: Mozart K. 331, mm 1-8

The activity here is to arrive at a description of the excerpt's phrases. The rules used in this example are shown in Table 1.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>K1</td>
<td>IF the goal is to analyze the melody and the phrase structure is not known THEN push a goal to describe phrases.</td>
</tr>
<tr>
<td>K2</td>
<td>IF the goal is to describe phrases and the melody has a key signature THEN push goal to find key and retrieve basic tonal information and retrieve information on that key signature.</td>
</tr>
<tr>
<td>K3</td>
<td>IF the goal is to describe phrases and the key is known THEN add that the phrase's cadence is a half cadence.</td>
</tr>
</tbody>
</table>

Table 1: KSM example rules

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>IF the goal is to describe phrases and the key is known and a phrase ends 7-2-1 THEN add that the phrase's cadence is an authentic cadence.</td>
</tr>
<tr>
<td>R2</td>
<td>IF the goal is to describe phrases and the key is known and the fourth measure ends 3-2 and the last measure ends on 1 THEN there are two phrases and label measures 1-6 phrase1 and label measures 5-8 phrase2</td>
</tr>
<tr>
<td>R3</td>
<td>IF the goal is to describe phrases and the key is known and there are two phrases and the first phrase has a half cadence and the second phrase has an authentic cadence THEN label the phrases as a period.</td>
</tr>
<tr>
<td>R4</td>
<td>IF the goal is to describe phrases and there is a period and the phrase beginnings are similar THEN label phrases as a parallel period and pop goal to describe phrases.</td>
</tr>
<tr>
<td>R5</td>
<td>IF the goal is to find the key and the key signature is 3 sharps and the last notes are C F A and the first note is a member of (A C E) THEN set the key to A major and retrieve a Major context and pop goal to find key.</td>
</tr>
</tbody>
</table>

Table 1 (cont'd.)

These rules are not presented as an ideal set for melodic analysis. Rather, they give the minimum knowledge base necessary to do the task at hand, and would be insufficient for other tasks. As of this writing the rules do not perform some basic actions such as counting measures in an example. These are currently handled by auxiliary functions but will be incorporated into the interpreter and rules in KSM's next revisions.

Given this set of rules, the melody, the information from the inventory functions, and an initial goal to analyze the melody, the following flow occurs in the program:

Fig. 1: KSM example control flow

Table 2 shows the changes in VM in this example:

<table>
<thead>
<tr>
<th>Step</th>
<th>VM ELEMENTS</th>
<th>RULE CHANGED</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Goal=analyze melody</td>
<td>R1</td>
</tr>
<tr>
<td>1</td>
<td>Goal=describe phrases</td>
<td>R2</td>
</tr>
</tbody>
</table>

Table 2: KSM example VM sketch
2 Goal-find key
3 Goal-describe phrases
   Keywords: major:
   Scale degrees: i, ii, iii, iv, v, vi, vii
4 (as above, adding:)
   Number of phrases: 2
   Phrase: \[\text{of-}\]
   Phrase: \[\text{of-}\]
   Last note: Phrase 2
5 (as above, adding:)
   Phrase: \[\text{of-}\]
   Phrase: \[\text{of-}\]
6 (as above, adding:)
   Phrase: cadence-mole
7 (as above, adding:)
   Phrase: structure-period
8 (as above, adding:)
   Phrase: period
   (end deleting)

Table 1 (cont'd)

In this example, K1 sets a new goal, which none of the appropriate rules can address until K2 has established the need to determine the key. K2 does this, as with the remainder of the rules continue until K7 has the last word, that
the phrases constitute a parallel period.

The flow of control here is partially explicit, in that two subgoals are set, and partially implicit, in that the (relatively) many rules with the common goal to describe the phrases are called according to the contents of Wm and the global conflict resolution scheme. In this middle level of the diagram, where the goal is describing the melody, K2 is chosen first as phrases have not been defined in Wm, K4 is chosen next as phrases are preceded by K3, K5 gives next, preceding K7 as there is still no mention in Wm of a period of any type. K7 compares this line of inquiry and returns control to its supragral, ending the chain of processing.

Rules are easily seen in the operation of this example; facts are used after their addition through the operations of K3 and K5. K5 adds a very simple invocation. The facts added by K5 include such changes as the information about scale degree which is used by K3, and K3, allowing for the translation from note-letters in the score to total abstraction in the rules. Thus mapping of detail onto generalization or abstraction is one typical use of facts in ESM, and allows for different levels of discourse and coarser or finer “granules” to be used when appropriate. In this way, rules can make use of generalities such as found in K3 and R3, with relatively few needing the highly specialized structures found, for example, in K8. Using the kind of knowledge found in K8, it makes many rules to compose the keys (see below typical of lower-level processing). This very fine grain is not necessary, but mandates a very large ruleset. The use of facts to provide values for constructs such as “interval” helps to avoid this.

Such a short example cannot hope to show all aspects of the system, but should give some feeling for how it operates. To date, ESM has been used for experimental applications in modeling problem-solving activities, in analysis, and in the design of tutorial environments.

6.2 EVALUATIONS

ESM is still a young system, but some lessons have been learned from it at this stage. The final two sections of the paper will summarize what appears to be the strengths and limitations of the PS approach and of ESM, and to show future directions for this research. Two main methodological points stand out:

1. Use different kinds of representation in the system, fitting each to appropriate tasks and knowledge. Structure the knowledge well.
2. Make control knowledge easily understandable by the user.

The circumstances under which each of these concerns has arisen, and the courses of action which they suggest, are detailed below.

1. Use different kinds of representation in the system, fitting each to appropriate tasks and knowledge. Structure the knowledge well.

The separation of knowledge in ESM into facts and rules seems to be a very useful technique, adding user-transparency without losing much in the way of uniformity of expression. The modularity of scales and facts allows for relatively rapid prototyping of a system, and for its incremental development. The addition of a score allows for music to be represented, facts allow for a proper mapping of detail into abstraction in dealing with the scores.

The structuring of rules and facts into hierarchies aids in the user’s comprehension of the system. The use of explicit goals in rules’ LHAs yields two benefits: less opaque processing in the system and simpler programming. The more visible view of control comes from the ability to find the same level of the system in pursuing the advantage to the programmer is not only to see real-time observation facility but also in the ability to partition the rule base in a natural way, grouping related rules together and creating hierarchical partitioning of the rule set. This allows rules themselves to begin to escape into the control realm where the overall guidance of the interpreter.

2. Make control knowledge easily understood by the user.

ESM provides a continuous of control possibilities to its user, from no set; control other than conflict resolution to the creation of explicit sequences of actions by careful goal-subgoal chaining. The hierarchical grouping of goals makes the development of larger, planned sequences easy, while allowing conflict resolution to happen in detail. If a great deal of detailed sequencing is needed, programming production systems can be preferable to a self-servicing solution, whether with ESM or another system.

ESM seems most useful in situations where a large
number of factors may enter into some problem-solving process. In these situations, the relatively "shallow" control structure of KSM is useful, allowing freedom to respond to any large number of occurrences with equal ease. In this, KSM acts as a prologue in which branching is the mainstay of program flow rather than being an exception, or where control is uniformly one "subroutine" (production) deep under the main program (macropage). In such instances, KSM is quite easy to use and works well.

7.0 FUTURE DIRECTIONS

KSM is still undergoing change and development. Three main areas in which the system may be improved are:

- Helping the programmer to add knowledge to the system and to organize it as well.
- Providing a user interface which helps to ameliorate the sparsity of a KSM operation.
- Implementing a satisfactory means of dealing with uncertain data.

These are discussed below.

Variety in which autonomous learning may best take place is one area in which KSM is being explored in an effort to ease the burden placed on the programmer in adding knowledge to the system. These include the automatic acquisition of new rules and facts and the creation of new contexts and new hierarchies of goal-groupings for rules. Considerable experimentation is necessary here, due to my desire to keep KSM reasonably deterministic, where results are reliable and repeatable. When systems Nos new knowledge, how it is spread throughout the system remains a key issue.

The user interface in KSM will also be revised to permit the programmer to directly input production operation and management. The excellent graphic capabilities of the Sun workstations in the Computer Music Studio at Northwestern make this desirable. Giving the user such features as a graphic rendering of the flow of control in the system or of the structure of the system and fact bases would be of great benefit in giving a more immediate apprehension of the workings of the system.

I have experimented with various means of dealing with uncertain data, including the use of "values" in rules to allow the system developer to specify the relative importance of some rule in some situation. The task KSM has been used for to date do not require this support at this time; it may be reintroduced at a later time should it be useful.

KSM will be of most interest in the near future in two areas: providing intelligent support for writing of music systems and in cognitive modelling. The former will be addressed in a later paper integrating the systems of the computer Music Studio at Northwestern. Of particular interest here is the possibility of altering the system to learn by observation, so that it builds up a pattern of behavior modeled on that of the system users. This is an extension of the two main uses to which KSM has already been put: modeling problem-solving activities and design of tutorial environments. In this way, the level of system support for the computer musician can grow invisibly and powerfully. The other area of main interest, cognitive modelling, is directed in particular to matters of cognitive control in music listening and to the development of expert-level listening skills. With the high degree of systems support in the Computer Music Studio at Northwestern, note progress is at hand.

REFERENCES


