Two papers on Musical Knowledge Representation

François Pachet
Laforia

e-mail: fdp@laforia.ibp.fr

abstract

Here are two papers about Music and Knowledge Representation, published in the International Computer Music Conference (1991, Montreal). The first one is an application of the NéOpus inference system to the analysis of jazz chord sequences. The second one is an architecture for representing knowledge used by Jazz Musicians while accompanying a soloist.

résumé

Nous regroupons ici deux papiers sur des problèmes de représentation de connaissances musicales, publiés dans les proceedings de l'International Computer Music Conference(1991, Montréal). Le premier est une application du système d'inférence NéOpus pour l'analyse de grilles d'accords de jazz. Le deuxième est une proposition d'architecture pour une représentation explicite des connaissances mises en jeu par les musiciens de jazz accompagnant un soliste.

A meta-level architecture applied to the analysis of Jazz chord sequences

François Pachet
LAFORIA, Institut Blaise Pascal,
Université Paris VI, 4, place Jussieu, 75252Paris Cedex 05

We describe here a knowledge-based system that analyses Jazz chord sequences, i.e. that find underlying tonalities for each chord of a given sequence. Rather than relying on formal properties of tonal music to perform the analysis, which yields good results only for a limited subset of well-formed chord sequences, we chose to mimic the human behavior, by representing the actual knowledge associated with this activity.
We show that the expertise involved in analyzing chord sequences consists both in domain knowledge (related to the theory of Standard Western Harmony), and meta-level knowledge, related to the control of reasoning. The system is based on a forward chaining, first order, Object-oriented environment, and performs the analysis by first looking for well-known patterns of chords (such as two-fives) for which the analysis is straightforward, and then by filling the gaps between analyzed patterns, using common sense knowledge, such as "prefer the analysis that minimize the number of underlying tonalities". The system is thus able to handle chords that would, in theory, be considered as incorrect (such as a Major seventh chord in first position in a blues).

Introduction. The problem

The problem of jazz chord sequence analysis consists in finding underlying **tonalities** for each of the **chord** of a given **sequence**. Chords are specified by their **root** (out of A, A#, B .. G#), and their **structure** (such as minor, major, major seventh, flat 5, 13). To simplify the presentation, tonalities are defined as **scales** (sequences of notes), and are indicated by their root and the **type** of the scale (out of major, harmonic minor, melodic minor). Analyzing a chord consists thus in finding the underlying tonality, and the **degree** of the chord in this tonality. An isolated chord may belong to various tonalities. For instance the chord [C Major] may be analyzed as (first degree of the C major tonality) or (fifth degree of the F major tonality). Only the relative positions of the chords in the sequence will allow deciding in which tonality the chord has to be analyzed.

A natural vs formal solution.

Two kinds of solutions exist for the problem. Formal solutions, such as grammar-based solutions [Steedman] consist in applying backward-chaining transformation rules to the sequence until a root sequence is found (such as the basic 12-bar blues sequence). Those methods rely heavily on the natural formal properties of chords, but are limited because they analyze only well-formed sequences that may be derived from a root sequence, which constrains heavily the number of sequences that may be analyzed. On the other hand, knowledge-based solutions consists in representing the knowledge used by musicians in the process of analyzing, and mimicking it in a appropriate environment. We explored the second solution, using the NéOpus Knowledge Representation environment [Pachet 1,3], in which knowledge is represented in terms of objects, forward-chaining rules, and meta-rules that control the reasoning.

A natural and general method

A common method for analyzing chord sequence is the following :

1- For each chord of the sequence, identify all the possible underlying tonalities, as given by the standard theory.

2- Identify well know small patterns of chords, whose analysis is straightforward, such as two-fives, or "turnarounds".

3- Once those patterns are found, fill the gaps between patterns, by using preference rules that maximize the extent of the underlying tonalities (or minimize the number of tonalities).

4- Use rules of thumb, that are not taken into account by the standard theory, such as "a seventh chord may be a first degree, if it is on a strong beat of the sequence", or "if there is a turnaround at the end of the sequence, then the underlying tonality of the turnaround is certainly the general tonality of the sequence".

In this scheme, several kinds of Knowledge have to be represented. We sketch out here how our system handle them.

An implementation with the NéOpus system

The NéOpus system is an Object-oriented, forward-chaining first order inference engine, inserted in the Smalltalk-80 environment [Pachet 1,3]. The Smalltalk-80 environment is well suited to represent domain objects, by their structure, and behavior. NéOpus adds a forward-chaining rule facility, that allow to represent knowledge in terms of relations between objects. Finally, a declarative control architecture allows to specify the control of knowledge bases by meta-rules.
In this system, the various kinds of knowledge are naturally represented : objects, rules and meta-rules.

Objects

Smalltalk classes represent the domain objects such as chords, tonalities, chord sequences. Thanks to the inheritance mechanism, common structure and behavior are shared. For instance, a root class called `ElementInASequence` is defined by having a `position` and a `length` in the sequence. Class `Chord` is defined as a subclass of `ElementInASequence`, that adds a `root`, a `structure`, and handles the tonalities. The chord sequence itself is represented by class `ChordSequence`, and is defined as having a `listOfChords` (instances of class `Chord`), and a `generalTonality`, as well as methods to handle access to its chords.
Class `Chord` implements methods to compute intervals, to perform transpositions, to compute the lists of pitches from the chord definition. Similarly, the computation of all possible tonalities for a given chord is handled

by Smalltalk methods, implementing the corresponding calculus. For instance, getting all the possible tonalities of a chord, say (A sharp, flat fifth, 11th) corresponds to the following Smalltalk message transmission :

```
(Chord !A# 5b 11) possibleTonalities.
```

The result will be a list (an `OrderedCollection`) of instances of class `Tonality`.

rules

NéOpus rules are used to represent the knowledge associated to patterns of chords. The syntax is the following sequence : a rule name, followed by a variable declaration part between vertical bars, in which variables are declared by their class name; a list of premises (which may be any Smalltalk expression) and an action part (also any Smalltalk expression) (see examples). Rules are used to represent knowledge associated with chord patterns, and chord analysis.

rules for defining patterns

For instance, the notion of "Two-five" is well represented in this manner. A "Two-five" is a sequence of two chords, such as (D min - G 7), so that : the root of the second is the root of the first transposed by a major fourth, and the first one is minor, and the second one major. However, a lot of variations on this basic definition exist. Once again, those variations cannot always be expressed in terms of transformation rules. For instance, the sequence (D 7 9+ - G 7) may be, in some contexts, interpreted as a two-five (if the surrounding chords have all been analyzed in the corresponding tonality for instance).

Here are for instance, two rules that define two-fives respectively in a major and a minor tonality :

```
majorTwoFive
    | Chord c1 c2 |
c1 isMinor.
c2 <- c1 followingChord.
(c1 includes: c1 flatFifth) not.
c2 isSeventh.
c2 root = c1 root fifth.
actions
|aTwoFive|
aTwoFive <-TwoFive new root: (c2 root fourth) index: c1 index; tonality: #major.
aTwoFive go
```

```
minorTwoFIve
    | Chord c1 c2 |
c1 isMinor.
c2 <- c1 followingChord.
c1 includes: c1 flatFifth.
c2 isSeventh.
c2 root = c1 root fifth.
actions
|aTwoFive|
aTwoFive <-TwoFive new root: (c2 root fourth) index: c1 index; tonality: #minor.
aTwoFive go
```

rules for performing the analysis

Rules are also used to represent the knowledge associated with the analysis itself. Those rules will find underlying tonalities for those chords who do not belong to analyzed patterns. General rules indicates that two adjacent, non analyzed chords that share only one common possible tonality have to be analyzed in this

common tonality. Other rules handle specific cases, such as the following rule: "when harmonic minor and melodic minor are possible for an isolated non analyzed chord, then chose harmonic".
Here are two rules that handle such cases. The first one says that the general tonality of a sequence is determined if there is a turnaround at the end. The second one chooses between harmonic and melodic tonalities in the case of a major isolated chord.

```
endOfSequenceAnatole
    | Turnaround a . Sequence s |
s == a sequence.
s generalTonality doesNotExist.
a isAtEnd.

actions
s generalTonality: a tonality
```

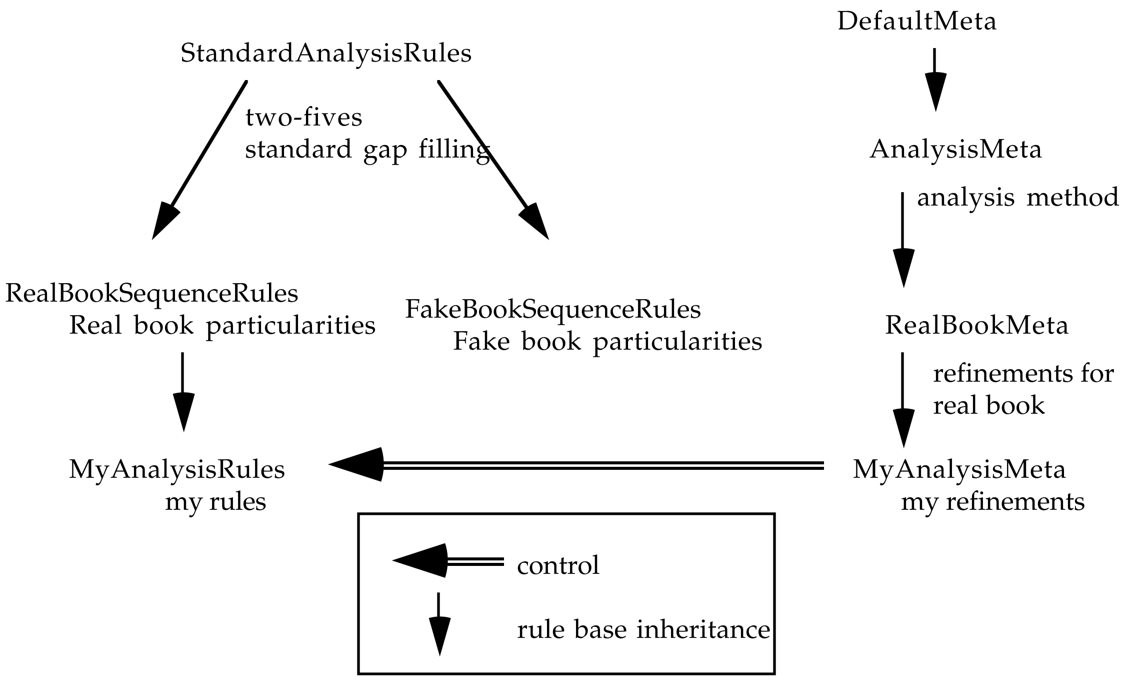
```
choiceMelHarm
    | Chord c |
c notAnalysed.
c isMajor.
c hasTonalityIn: #minorHarmonic.
c hasTonalityIn: #minorMelodic.
actions
c analyse:
     (c tonalityIn:#minorHarmonic).
```

meta-rules

Rules are not to be triggered in any order. The process of analyzing consists in various steps that are not reflected in the rules themselves. In order to represent this, we use the control architecture of NéOpus, to specify the control of those rules. Rules are organized in protocols (groups of rules). In order to stick to the natural method presented above, packs of rules have to be executed in a particular sequence.
Our architecture separate the domain rules, from the control rules. Rules are organized in rule bases, which may be activated by other rule bases, called meta-bases. In our case, all the domain rules (pattern identification, gaps filling) are included in rule-base `Analysis`. A meta-base `AnalysisMeta` contains all the rules necessary to represent the chosen analysis method. `AnalysisMeta` controls the activation of `Analysis` by an association handled by the system. Other meta-bases are implemented, that define other analyzing procedures, and may be associated the `Analysis` rule base.

rule base inheritance

Rule bases may interact with each other, using a rule base inheritance mechanism, translated from the class-based language inheritance mechanism, with a different interpretation: a rule base A inheriting from rule base B (defined as a sub-base of rule base B), will be considered more specific than B, and all its super bases. Rules of the most specific rule bases will be preferred at execution time, by the rule interpreter. A rule base tree is thus constructed, in which rule bases implementing more specific knowledge will inherit from more general rule bases. In our example, two different branches are built, one for domain rules, from root base `StandardAnalysisRules`; the other for control strategies, from root base `DefaultMeta`. More specific rule bases are defined, that handle the variations necessary to analyze sequences from the Real book, or the Fake book. Then user's rule bases are defined as sub bases of such existing rule bases.



the rule base inheritance tree for chord sequences analysis

Discussion

Our system is to be compared with the backward-chaining approach described in [Steedman], that defines a set of transformation rules with which an initial chord sequence is analyzed and reduced to a root sequence (such as the 12-bar blues). We are mostly interested by those chord sequences that would not be considered syntactically correct by such grammars, but for which there are acceptable solutions (the evidence is that musicians do analyze such chord sequences). For instance, tunes like C. Mingus's "Good bye pork pie hat", or even the standard D. Gillespie "Night in Tunisia" may not be analyzed correctly with those techniques, because they are only locally syntactically correct, and do not derive from a root sequence. Our system gives plausible analyses for those tunes. However, unlike backward-chaining systems, it is not able to give multiple solutions in cases of ambiguous chords. Moreover, our system is interesting because it gives the opportunity of experiencing various control strategies on the same knowledge base.

This system is an application of a general architecture for representing deductive knowledge in an Object-Oriented environments [Pachet], which has also been applied for representing Knowledge used by accompanying Jazz Musicians.

References

[Pachet F. 1] NéOpus User's Manual. Rapport LAFORIA, n° 91/14, July 91.
[Pachet F. 2] Towards a expert system that follows human improvisation. Rapport de DEA. Ircam, Paris 1988.
[Pachet F. 3] Reasoning with objects : the NéOpus environment, East EurOOpe, Bratislava, Sept. 91.
[Steedman Mark J.] A Generative Grammar for Jazz Chord Sequences. Music Perception, 1984, Vol. 2, N° 1, pp 52-77.

Representing Knowledge Used by Jazz Musicians

François Pachet
LAFORIA, Institut Blaise Pascal,
Université Paris VI, 4, place Jussieu, 75252Paris Cedex 05

Abstract : Our system is a framework for building Expert Systems that simulate jazz musicians 's playing. The music is generated according to a given jazz chord sequence, and the real-time MIDI input of a soloing musician. A MIDI output is generated, which control MIDI synthesizers. The style of music generated is standard "real book" style swing jazz. We outline here the basic architecture of this system, which relies on a three axis representation scheme : objects, rules and strategies.

Introduction. What kind of Knowledge ?

We describe here an architecture that allows representing certain kinds of Knowledge used by accompanying Jazz musicians, namely swing bass lines or piano comping. The task of writing such knowledge-based systems is made extremely difficult by the diversity of the knowledge which varies from very formal definitions (inherited from Classical Music) such as the formal theory of chords, modes, scales, to largely heuristic principles ("play the tonic of the chord in the first beat of the measure" for a bass player).

Procedural/Declarative Knowledge. Knowing how to build or analyze a chord, play a scale, find out common pitches for a given set of chords, may be considered procedural in the sense that there are algorithms available that do the job. More complex operations such as analyzing a sequence of chords would not be considered procedural [Pachet 91]. Similarly, adding superstructures to a chord would not be considered procedural as it relies heavily on the context, which may be very complex. Moreover, declarative Knowledge in Music seems to be essentially non-monotonous and uncomplete. Typically the bassist leitmotiv *"play the tonic on the first beat"* is uncomplete (expliciting the precise contexts in which this is true or false is impossible) and non-monotonous (the truth value of this statement may vary over time, chord changes, context).

Implicit/Explicit Knowledge. Representing Knowledge is limited to representing only explicit knowledge. Our system therefore represents abstractly the ideal "rational agent" for which every action taken has justifications that are coherent for its inference mechanism.

Importance of licks. The activity of live playing is not entirely spontaneous and creative. A tremendous corpus of information acquired through experience is used, retrieved, modified along with inference activities. This corpus consists in pre-existing sets of musical phrases, chords, sequences of chords associated with (uncomplete) contexts of validity. This corpus play a central role in live music. It may even constitute a characterization of a player's style (i.e. the licks that recurrently appear in its playing). A representation of this somehow rigid knowledge has also to be set up.

Role of Time . Any musical component has a time dimension, which has some properties used by the inference : the sequenciality of objects, duration, metrics, are all relevant aspects to be taken into account by the system. But time plays also an important role with regards to expectation/failure. Musicians play with the expectation of the audience by fulfilling expectations or creating ruptures or breakdowns. To handle this, an history or trace of what has been played must be used and structured so as to be usable by the inference system.

Description of the system

According to the previous requirements, our system is build around a triadic paradigm : objects, rules and strategies. Those three elements form the kernel of our representation language and are articulated so as to provide the expressive power needed. They are handled by an inference engine which produces the music (a Midi output) according to the Midi input, the chord sequence and all the Knowledge in its various forms. Time is not explicitly represented but is handled by the engine in an ad'hoc way. Music in input is transformed into structured objects (notes and chords) having absolute time values. The engine produces music by chunks, and constantly checks its remaining time (i.e. the time left before the expected beginning of the chunk). A Midi driver [Boynton 86] is in charge of producing Midi output according to their time tags.

Time management. A distinction is made between the *logical* time and the *real* time. The real time is the actual time as given by the processor clock in milliseconds. The logical time is the time as handled by the engine and can be seen as "what the system is currently thinking of". This distinction is crucial, as the engine will constantly check it has enough time to spend on thinking in order to choose between alternatives : costly calculations will be conducted only if it has enough remaining time. Conversely, if the engine is late or "behind the schedule" (i.e. its logical time is greater than the real time) then the engine will chose direct "stimulus-response" like actions (licks for instance) in order to make up.
This criteria is explicit in the system (there is a rule that dispatch actions according to the difference between real and logical time). The tradeoff between the time taken to chose an action and the effective time taken to perform the action is a constant preoccupation in meta Knowledge Representation[Pitrat 90] and is given here an explicit representation.

Objects. Musical objects are structured according to the Object-orientation paradigms (i.e. instanciation, inheritance, message passing). In this scheme, classes are defined which describe both structures and behavior of their instances. For instance, chords are represented as instances of the class `Chord`, having attributes `listOfNotes`, and `harmonicDegree`. This provides an excellent platform to implement various procedural Knowledge such as :

- structural knowledge. Objects are structured (in terms of attributes) and thus constitute a network : a chord sequence is composed of chords, themselves composed of a list of notes and a duration. A note consists in a pitch and a velocity and so on.
- Procedural Knowledge associated with these Objects. A chord for instance will be able to compute inversions, transpositions, to extract its most important pitches. Those procedures are called methods.
- Handle multiple representations. For instance, a musical phrase may be described as a sequence of notes, themselves described as having a pitch (a number, an initial time, an ending time, a velocity (or pressure)). Such a representation is inadequate at music production time : a representation using durations instead of absolute time will be preferred. Changing the representation will be made via a method in class `MusicalPhrase`.

Moreover, Objects are also used to structure the environment itself : A class `BassPlayer` is defined, having an **hear** (the object in charge of listening to the input data and converting it into a readable form), a **memory** (in charge of managing the various data containing the pre-recorded licks), a **mind** (the actual inference engine and the rule base), an **eye** (which sees and handles the chord sequence).
This structuring provides a natural representation of the Musician itself, and thanks to inheritance, makes specialization of these classes easy. For instance a class `PianoPlayer` will share a lot of common functionalities with the class `BassPlayer`. Only minor changes have to be made, so both classes will inherit from an abstract class `Musician`, and will define only those characteristic behaviors.

Rules. Knowledge related to groups of objects and their interactions is represented by rules. These rules implement the various operations a Musician can perform. These operations are well represented by rules because they include a context part (the definition of the context in which the action may be performed) and an action part (the actual action to be performed). The classical if-then production rule is thus adequate. However, the contexts of activations are not entirely defined : those rules act more like suggestions or advice than orders. Very often, those contexts are empty. For example, "the bass can play an arpeggio starting with the tonic on the first beat" may be represented as a rule in the system, with an empty context (a bass player can *always* do this).
Rules are organized in packs, each pack representing a certain chunk of Knowledge. The packs are not necessarily independent and may share objects. The packs may themselves be recursively structured in packs.
For the bass player rule base, a typical rule may thus be written as follows, stating that a possible action is to produce an arpeggio from the root of the current chord, in the same tonality :

rule	r
	(currentChord ?c ?- ?- ?tonality)
then	(produce (arpeggioFrom ?tonality ?c))

This rule would be written in the pack "arpeggios", as well as other rules dealing with the many variations around producing an arpeggio. The conclusion part of the rule adds a new "`produce`" fact in the fact-base, with the necessary information to actually compute the musical data. This `produce` fact will be handled by a specific pack of rules that will, if this action is actually chosen, eventually produce the corresponding Midi output data.

Strategies. As in any forward-chaining mechanism, rules have to be chosen, according to the actual contexts. Those choices involve an other kind of Knowledge called meta-Knowledge [Pitrat 90]. In order to provide a conceptual framework to represent this knowledge, we introduced the notion of strategy : a strategy is any potential action related to the music to be produced. It is structured as :
a `type` (specifying the nature of the strategy), a `time interval validity` (possibly open), a `name` (the name of the action), a `set of rules` (actually implementing the strategy).
Typical strategies will be : "play louder", "play more notes", "play arpeggio", "don't play", "play atonal", "play lick number 7612", "play last played phrase transposed by a half step" ...
Strategies being defined as classes, as for domain objects, subclasses of `Strategy` can be defined, having particular behavior, or adding structures. For instance, strategies of a special kind (say `OrderedStrategies`) may have priorities that will be used at conflict resolution time.
Since several strategies may be valid at a given time, strategies will interact with each other, at conflict resolution time, in order to determine the final decision for output. Conflict resolution will also be handled by rules, which will decide according to the current context which strategy to chose. In particular, this scheme allows the notion of "long-term" plan to be (partially) represented : once a strategy is chosen, the corresponding pack of rule is triggered, and the cycle starts again. At the next selection time, the strategy may still be valid. The choice of keeping the old strategy or choosing a new one will be made by conflict resolution rules.
Strategies are not always exclusive. For instance a strategy like "play louder" may coexist with a strategy like "play an arpeggio". The strategy "play louder" will not actually produce musical output, but will simply modify the musical data produced by the strategy "play an arpeggio", by producing a louder arpeggio. This distinction is represented by the attribute `type`. Strategies that actually produce something will have the type #`produce`, whereas strategies influencing others will have the type #`influence`. Of course type is not restricted to those two values, and other types can be added.

The Engine cycle. The engine cycle is classically threefold : identification of all matching rules, selection of one rule, application of the action part of the rule. The management of strategies is entirely described by rules. Rules are structured in packs, and each pack is triggered by the action specified in the action part of a rule. Rules implement the strategy management in three steps :

- identification of all possible strategies,
- conflict resolution : choice of actual strategies
- music production : according to the various strategies chosen, musical data is generated.

The top level rules for instance looks like :

```
rule   rLate
if
       (currentTime ?t)
       (logicalTime ?l)
       (< ?t ?l)              ; we are late on the schedule
then   (trigger fastDecisionPack)
```

a similar rule will handle the case when there is enough time to think :

```
rule   rAhead
if
       (currentTime ?t)
       (logicalTime ?l)
       (> (?t (+ ?l 1000))      ; we are 1 second ahead on the
schedule
then   (trigger slowDecisionPack)
```

The pack `fastDecisionPack` will consist in choosing pre-recorded licks (in the `memory` of the Musician). Strategies themselves are handled by the lower level packs.

Current Implementation.

The current representation uses the n-ary predicate syntax. In this scheme, facts of the fact base are described as lists whose first element is the predicate's name, and the remaining are arguments. Arguments may themselves be the result of Lisp functions (written between parenthesis).
Strategies are therefore represented as : `(strategy aName firstBeat lastBeat aPackOfRules type)`. The chord sequence is represented by chunks : only the current chords are present in the fact-Base. The chord's structure is `(currentChord aChordName firstBeat lastBeat underlyingTonality)`, where `aChordName` is the chord's name according to the chords naming conventions; `firstBeat` and `lastBeat` are the current chord's range of validity in the chord sequence, and underlyingTonality is the current tonality. An implementation in Smalltalk-80 using the NéOpus [Pachet 91] environment is in progress.

Discussion

This representation scheme allows for representing a great deal of what a Musician (actually the author) can say about its own playing. It provides a good framework to differentiate between different types of Knowledge. Objects provide a very good and flexible representation framework. Rules give a good representation for potential actions to be performed, independently of their contexts of activation, and strategies, acting as an abstract representation of those rules, allow to separate between the description of actions, and the decisions related to those actions. However all Knowledge is not well represented. Notions related to time are poorly handled. A representation of rhythm is essential here, and should be described as a component of an other object. Our system does not have the ability of handling rhythm as a first class object. It is accessible only through the time properties of the objects. The level of abstraction is not sufficient now in this concern, and such notions as `swing`, *on the beat*, *slightly ahead*, are not easily represented (they are accessed only by costly and hazardous computations).

References

[Boyton L] preFORM : an Object-Oriented environment for Music Composition. ICMC '89.

[Boyton L & al.] Midi-Lisp, a Lisp based Music Programming Environment for the Macintosh. ICMC '86.

[Jackendoff R., Legdahl F.] A Generative Theory of Tonal Music. Cambridge, MA. MIT Press, 1983

[Pachet F.] NéOpus User's manual. Rapport LAFORIA, July 91.

[Pachet F.] Towards a expert system that follows human improvisation. Rapport de DEA. Ircam, Paris 1988.

[Pachet F.] A Meta-level Architecture for analysing Jazz chord sequences. Same conference.

[Pitrat J.] Métaconnaissance, Hermes, Paris 1990.