The MusiES system: an environment for experimenting with knowledge representation techniques in tonal harmony

FRANÇOIS PACHET.

LAFORIA, Institut Blaise Pascal, 4, Place Jussieu, 75252 Paris Cedex 05, France

Abstract

We report here on current works on the MusiES environment, designed for experimenting with various object-oriented knowledge representation techniques in the field of tonal harmony. The first layer of MusiES is a repository of consensus knowledge about tonal harmony, including an explicit representation of enharmonic spelling, as well as representation of intervals, scales and chords that support standard computations. We give an overview of several systems built on top of MusiES: a system for the analysis of jazz chord sequences, a system for the automatic generation of harmonizations, and a system that generates real-time jazz improvisations. We give an overview of MusiES and its extensions and discuss several representation issues and their solutions in MusiES.

1. Introduction: yet another Smalltalk music representation system

Music analysis has long been a favorite domain for researchers in Artificial Intelligence. Within AI, Object-Oriented Programming (OOP) has traditionally been a favorite paradigm to build complex musical systems, especially so oriented towards synthesis (from the Forme system (Covent&Rodet 1991) to the MODE system (Papa 1991), the Kyma system (Scartlet 1987), and ImprovisationBuilder (Walker and al., 1992)). Of course, object-oriented programming has been - and is used - for almost any kind of complex software, but there is hardly no mention in the literature of attempts to use specifically OOP techniques to implement analysis systems in tonal music. Following this tradition, we are interested in developing intelligent systems specialized specifically in tonal harmony, using object-oriented techniques. This paper is a report on our results so far, embodied in the MusiES system.

From a totally different point of view, our main object of study is the construction of large knowledge bases, using OO techniques, and Smalltalk in particular. In this context, tonal harmony is seen as an ideal field for obvious reasons: it is complex yet understandable, involves complex structures which call for non-trivial representations (e.g. intervals, chords), requires adequate representations of time, involves abstract notions (such as analysis, degrees), and so forth. In this respect, we are interested in the integration of OOP with various inference mechanisms to produce truly reusable knowledge-based components.

We will first briefly describe the foundation of MusiES, dealing with pitch classes and basic concepts of tonal harmony, then give an overview of three systems built on top of MusiES, and end up with a discussion of the results.

2. Representation of pitch classes

2.1. The problem?

One of the foundations of the MusiES system is the representation of pitch classes (hereafter referred to as PC), that respects enharmonic spelling (i.e. the difference between notes that spell differently but sound the same, such as Eb and D#). Enharmonic spelling is as vital to music analysis, as orthography is to grammar and semantics. Although it may seem a remarkably simple problem, it has, to our knowledge, yet never been fully addressed. For instance, Wiegand (92) emphasizes the importance of taking enharmonic spelling into account, but proposes an ad hoc representation of chords as Lisp dotted lists. Similarly, Steedman (84) proposes a solution for performing harmonic analysis of chords sequences but, considers all the entities (chords, intervals or notes) as Prolog-like constants and is interested only in higher level properties of sequences deduced from the mere ordering of their elements. More generally, MusiES addresses the problem of providing a "good" representation of the algebra of pitch classes, including the notion of "enharmonic spelling", and a representation...
of intervals, scales and chords to serve as a foundation for implementing various types of harmonic analysis mechanisms. This calculus must take into account various facts and properties of pitch classes, such as:

- There are conceptually 35 different PCs: 7 naturals, 7 flats, 7 sharps, 7 double sharps and 7 double flats, with only one occurrence of each PC (in our octave-independent context). Practically, this means that, for example, the minor second of B (C) is physically the same note as the minor seventh of D, and so on.
- PCs are linked to each other half-tone or tone wise, and form a circular list. But some notes are pitch-equivalent, (e.g. A# and Bb, or C#, D and Eb).
- There is an non trivial algebra of alterations, which includes the following pseudo-equations:
  \[
  \#0 = 0 = \#0 = \text{identity}.
  \]
  For any x in (0, natural), x * 0 = natural.

This algebra is non trivial because not everything is allowed, at least in the classical theory, e.g. triple sharps.

- PCs are linked by the notion of interval, which, in a way, preserves this algebra. For instance, the diminished fifth of C is not the same PC as the augmented fourth of C, but both PC sounds the same.
- Certain intervals are forbidden for certain PCs: for example, the diminished seventh of C# does not exist (it would be B bbb).
- Certain scales do not exist, by virtue of the preceding remarks: G# major is impossible (because it would contain a F# in its signature). The same holds for D# harmonic minor, and so on.

2.2. Pitch classes as instances

Although it is possible to write a global algorithm in any procedural language that takes all these cases into account, there is clearly here a better solution, which consists in treating pitch-classes as instances of classes, in the sense of OOP, and alterations as methods for these classes, using polymorphism to represent their algebra, and all the properties mentioned above. This approach not only yields a simple implementation, but also provides us with a clear understanding of the operations on pitch classes.

We define 5 different types of pitch-class classes (to avoid long names, we refer to pitch-classes as "Note"): PitchClass, PitchClassFlat, PitchClassSharp, PitchClassDoubleFlat and PitchClassDoubleSharp. Distinguishing between different classes for pitch classes gives us a precise definition to alterations: the #, b, and natural, are then represented as polymorphic methods on these classes. For example, the # operation maps instances of PitchClassNatural to instances of PitchClassSharp, and A# is then seen as the result of operation # to A.

This operation is polymorphic because there are actually four distinct sharp operations, depending on the class of the argument. In order to represent notes according to these requirements, we define a hierarchy of classes as follows, where each class defines its set of instance variables and operations: 

- PitchClass represents the root of all classes representing pitch-classes. It is an abstract class and has no instance variables.
- PitchClassNatural represents natural pitch-classes. There are 7 instances of PitchClassNatural, representing the 7 natural notes (A, B, C, D, E, F, G). They are linked to each other according to the order (A, B, C, D, E, F, G), and have two pointers on their corresponding sharp and flat PC. 
- PitchClassNatural is the root of the classes representing altered (and doubled altered) notes. It defines only one instance variable (natural) pointing back to the natural note it comes from (e.g. A#, A, #B, Ab, and Aflat all have A as their natural). Finally, there are four subclasses of PitchClass for representing respectively sharp, flat, double sharp and double flat notes. These classes implement the methods sharp, flat and double flat so as to respect the natural algebra of alterations. As an example, here is the list of all the implementations of method flat:

<table>
<thead>
<tr>
<th>PitchClassNatural methodsFor: 'alterations!'</th>
<th>PitchClassFlat methodsFor: 'alterations!'</th>
</tr>
</thead>
<tbody>
<tr>
<td>flat</td>
<td>flat</td>
</tr>
<tr>
<td>flat</td>
<td>flat</td>
</tr>
<tr>
<td>flat</td>
<td>flat</td>
</tr>
<tr>
<td>natural</td>
<td>natural</td>
</tr>
</tbody>
</table>

Note that the flat operation is intentionally not defined for class PitchClassDoubleFlat. The flat message sent to a PitchClassDoubleFlat will raise an error, which is consistent with our philosophy. The same patterns applies for method sharp in PitchClassDoubleSharp, as well as for most operations in pitch classes (computation of intervals, of semitones distances, transpositions, etc.)

### 3. Basic Harmony

Once pitch classes are correctly represented, we add the representation of all the basic concepts of tonal harmony, including octave-dependent notes, intervals, scales (classical and exotic ones), and chords. These classes and methods were designed to support basic computations such as the one taught in first year courses. For reasons of space, we will not discuss their representation here (see Pachet (1994) for details), and simply give a few examples of what the system can do.

#### 3.1. Alterations on pitch classes and OctaveDependentNotes

A bunch of methods represent most common computations on pitch classes and octave dependent notes, to compute alterations, and test pitch equality, such as:

- PitchClass C -> C# "PCs are accessed by class messages"
- PitchClass C sharp -> C# "C#"
- PitchClass C sharp sharp flat -> C# "C#"
- PitchClass C flat flat flat -> Error: "flat" not understood by class DoubleFlatNote
- PitchClass C sharp equals: Note D flat -> true "true"
- PitchClass C sharp octave: 3) sharp -> C#3 "an OctaveDependentNote"

#### 3.2. Intervals

Intervals are represented as first class objects. Methods allow their creation from notes, and their manipulation is any possible way. Here is an excerpt of methods dealing with intervals:

- PitchClass C flatFifth -> Gb
- PitchClass C augmentedFourth -> F# A
dd
- PitchClass C majorThird majorThird -> G# A#
- PitchClass B sharp octave: 3) fifths -> E4
- PitchClass C flat diminishedSeventh -> error: illegal interval
- Interval diminishedFifth bottom1Tops: (PitchClass F sharp) -> C
- Interval diminishedFifth bottom1Tops: (PitchClass C flat) -> C Db
- Interval majorThird reverse -> minor sixth
- Interval perfectFifth + Interval majorSecond -> majorSixth
- PitchClass C intervalWith: PitchClass F sharp -> augmented fourth

#### 3.3. Scales

ManES provides a representation of scales that allows easy computation of derived modes, and derived scale-tone chords. Adding new exotic scales is done by defining new subclasses of class Scale, with corresponding interval lists.

- PitchClass A flat majorScale -> Ab major
- PitchClass A flat majorScale notes -> (Ab Bb C Db Eb F G)
- PitchClass C harmonicMinorScale notes -> (C D Eb F G Ab B)

#### 3.4. Chords

Chords are an important - an complex - concept in tonal harmony. ManES provides a complete vocabulary that allows to name and manipulate all possible chords (from 2 to 7 notes). Chords may be represented either by their name (a string), or from a list of notes. Here are some examples of chord name computations using both mechanisms:
representation of chords, and involve very specific representations that are irrelevant for simple monophonic melodies. A trivial connection to MIDI has been realized, using Bill Walker's MIDI Smalltalk primitives for the Macintosh. Since this is not our priority, only basic (but useful enough) play functions have been implemented, and no sophisticated recording (and hence quantization) is realized in the current version.

4.1. Graphical Editors for melodies

Graphical score editors are not only useful for our purposes. They are, in a way, a particular knowledge bases, incorporating lots of knowledge about musical notation. For instance, the problem of knowing which way to draw beamed (up or down) and how to group eighth or sixteenth notes together, how to split notes whose durations exceed certain amounts of time (synopses), where to position notes and so forth, are problems that do require lots of musical knowledge to be solved. We started to implement a series of graphical editors (Cf. Figure 1) for both monophonic and polyphonic melodies, with standard edition operations (key transpositions, copy/paste, file/import, etc.) For the moment these editors are written in Smalltalk. However, we plan to redesign them using more declarative knowledge representation mechanisms (constraints), and integrate them in a uniform system about musical notation.

5. Extensions

MaStS is used (and validated) by several knowledge-based system built on top of it. We give an overview of three of them. More details can be found in the references.

4.1. Project #1: Analysis of Chord Sequences

The first project is the construction of a knowledge-based analyzer for jazz chord sequences. The sequences are standard be-bop tunes as found in the Real Book/Fake book corpus. The aim of this system is to find underlying regularities for each chord in a sequence, when possible. Previous approaches to this problem mostly based on a formal theory of the underlying domain. For example, Steedman (84) uses context-dependent generative rules to model 12-bar blues, that capture all "legal" distortions from the original 12-bar blues sequence. However, our model is not directly implementable, and yields solutions only for well formed chord sequences. On the contrary, our system is based on a model of the reasoning as it is made by experts, and is divided in two phases: 1) pattern recognition in which the expert "sees" particular well-known shapes, whose analysis is trivial, such as Two-Five's, Two-Five-One's, Turnrounds, resolutions, etc. and 2) gap filling phase, in which isolated, non analyzed chords are grouped to adjacent analyzed shapes when possible.

The system is an extension of MaStS with classes to represent chord sequences and objects used for the analysis (the shapes, the analysis themselves, etc.). The reasoning is represented by rule bases expressed in NCOps, a first-order forward-chaining inference engine integrated with Smalltalk-80 (Cf. Pachet (96)). The model of the reasoning is described in depth in Pachet (1994b) and Pachet (1991), and uses a declarative architecture for representing control knowledge (Pachet & Pernet (1994)).

4.2. Project #2: Constraint satisfaction and automatic harmonization

This system is an attempt to capture musical rules as found in treatises of harmony and counterpoint. These rules are most often stated as constraints, such as "the interval between two successive notes in a melody should be consonant". One of the major drawbacks of the previous attempts (Ebertioglu (1991), Chen (1991)) is the independence of the constraint satisfaction mechanism, leading to inefficiencies and complex knowledge bases. The aim of the system is to explore the integration of constraint-satisfaction mechanisms (arc-consistency) and intelligent search (branch & bound), with our existing object structures. This work is still in progress and shows already very promising results in terms of efficiency, compared to previous attempts by Chen (1991) and Ballam (1994).

5.3. Project #3: Simulation of real-time jazz improvisations

This system is an attempt to build a musical memory that explains - at least partially - improvisation processes. A model of memory, based on case-based mechanisms (cf. Ramalho & Ganaccia (1994) has been developed, and is used in conjunction with a representation of musical actions or PACTS (cf. Pachet (1991),...
advantage of being simple to implement, but suffers from the other drawbacks. For instance, information such as the “following” note in a melody is not easily accessible (in either of the representations). We are currently investigating a more convenient representation of time, based on the extensive use of “wrappers”.

6. Conclusion, future works

We described MusiES, a knowledge base that represent concepts of basic harmony and their most current operations. We gave an overview of three systems built on top of MusiES, that use MusiES structures in conjunction with various inference mechanisms. Future works include 1) The connection of the graphical editors with MusicWriter, an extension of LaTeX for musical scores, to generate professional quality scores from our editors, 2) The representation of pitch-classes using the two-dimensional Harmony Space interface described in Holland (89); 3) Continue with the representation of musical rules (counterpoint of simple species), as well as Schenkerian analysis; and 4) Using MusiES and its extensions as a tutorial system.

7. References


