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

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

Abstract

We report here on current works on the MusES environment, designed for experimenting with various object-oriented knowledge representation techniques in the field of tonal harmony. The first layer of MusES 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 MusES: 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 MusES and its extensions and discuss several representation issues and their solutions in MusES.

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 oriented towards *synthesis*, from the *Form*es system (Cointe&Rodet 1991) to *MODE* (Pope 1991), *Kyma* system (Scaletti 1987)), and more recently *ImprovisationBuilder* (Walker & al., 1992). Of course, object-oriented programming has been - and is - used for almost any kind of complex software. However, there is hardly any mention in the literature of attempts to use specifically OOP techniques to implement analysis systems in tonal music. Therefore our endeavor to develop intelligent systems specialized in tonal harmony, using object-oriented techniques, appears to be new. This paper is a report on the results we have obtained so far, which are mainly embodied in the MusES system.

From a methodological point of view, our main object of study is the construction of *large knowledge bases*, using object-oriented techniques, and Smalltalk in particular. In this context, tonal harmony is seen as an ideal field for obvious reasons: it is complex yet understandable, it involves complex structures that call for non-trivial representations (e.g. intervals, chords, etc.), it requires an adequate representation of time, it involves abstract notions (such as analysis, degrees, etc.), 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 MusES, dealing with pitch-classes and the basic concepts of tonal harmony, then give an overview of three systems built on top of MusES, and end up with a discussion of the results.

2. Representation of pitch-classes

2.1. The problem?

One of the foundations of the MusES system is the representation of pitch-classes (hereafter referred to as PC), that conforms to 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, Winograd (93) 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 where he 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, MusES 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 B*b* , or C##, D and E*bb*).
- There is an non trivial algebra of alterations, which includes the following pseudo-equations:

  # o *b* = *b* o # = identity.

  For any x in (#, *b*, natural), x o natural = 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 sound the same.
- Certain intervals are forbidden for certain PCs: for example, the diminished seventh of C*b* 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*b* 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 name, we refer to pitch-classes as "PC"): `PCNatural`, `PCSharp`, `PCFlat`, `PCDbleFlat` and `PCDbleSharp`. 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 `PCNatural` to instances of `PCSharp`: A# is then seen as the result of operation # to pitch-class 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. `PCNatural` represents natural pitch-classes. There are 7 instances of `PCNatural`, 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. `PCAltered` is the root of the classes representing altered (and doubly altered) notes. It defines only one instance variable (*natural*) pointing back to the natural note it comes from (e.g. A#, A##, A*b* , and A*bb* all have A as their *natural).* Finally, there are four subclasses of `PCAltered` for representing respectively sharp, flat, double sharp and double flat notes. These classes implement the methods `sharp`, `flat` and `doubleFlat` so as to respect the natural algebra of alterations. As an example, here is the list of all the implementations of method `flat`:

!PCNatural methodsFor: 'alterations'! **flat** ^flat	!PCFlat methodsFor: 'alterations'! **flat** ^flat
!PCSharp methodsFor: 'alterations'! **flat** ^natural	!PCDbleSharp methodsFor: 'alterations'! **flat** ^natural sharp

Note that the flat operation is intentionally not defined for class `PCDbleFlat`. The flat message sent to a `PCDbleFlat` instance will raise an error, which is consistent with our requirements. The same pattern applies for method `sharp` in `PCDbleSharp`, 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 harmony courses. For reasons of space, we will not discuss their representation here (Cf. 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#	
PitchClass C sharp sharp flat	-> C#	
PitchClass C flat flat flat	-> Error:	'flat' not understood by class DoubleFlatNote
PitchClass C sharp pitchEquals: Note D flat	-> 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 in any possible way. Intervals may be used explicitly (as in the two first examples) or indirectly via methods attached to pitch-classes. Here is an excerpt of methods dealing with intervals:

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

3.3. Scales

MusES 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. MusES provides a complete vocabulary that allows to name and manipulate all possible chords (from 2 to 7 notes). Chords may be created either from their name (a string), or from a list of notes. Here are some examples of chord name computations using both mechanisms:

```
(Chord new fromString: 'D# maj7') notes              OrderedCollection (D# F## A# C## )
(Chord new fromString: 'C 13 aug9 no7') notes   OrderedCollection (C E G D# F A )
Chord newFromNoteNames: 'C E G'                      [C]
Chord new FromNoteNames: 'C F G'             [C sus4]
Chord newFromNoteNames: 'C E F F#'           [C no5 no9 no7 11 aug11]
Chord newFromNoteNames: 'D# F## A C##)'      [D# dim5 maj7]          "from A. Holdsworth"
```

Similarly, there are methods to compute the list of all *plausible* chord names for a list of notes:

```
"the root is one of the notes: "
Chord allChordsFromlistOfNoteNames: 'C E G'
->  OrderedCollection ( [C] [E min no5 no7 no9 no11 dim13] [G sus4 no5 6]

"the root is any note, possibly not in the list:"
Chord reallyAllChordsFromlistOfNotesNames: 'C E G'
->  OrderedCollection ([A noRoot min 7 ] [B noRoot sus4 no5 no7 dim9 dim13 ] [C ] [D noRoot sus4 no5 7 9 ]
[E min no5 no11 no9 no7 dim13 ] [F noRoot no3 maj7 9 ] [G sus4 no5 sixth ] [A# noRoot no3 dim5 ] [C#
noRoot min dim5 ] [D# noRoot no3 no5 dim9 ] [F# noRoot no3 dim5 7 dim9 ] [G# noRoot no3 no5 no11 no9
no7 dim13 ] [Ab noRoot aug5 maj7 ] [Bb noRoot no3 no5 no7 9 aug11 sixth ] [Db noRoot no3 no5 maj7 aug9
aug11 ] [Eb noRoot no5 sixth ] [Gb noRoot no3 no5 no9 no7 aug11 ] )
```

An important notion is the notion of "scale-tone chord", extracted from scales by successive thirds. The following expression yields all 4 voice scale-tone chords generated from the scale (C Hungarian Minor):

```
(HungarianMinor root: PitchClass C) generateChordPoly: 4         ->
OrderedCollection ([C min maj7] [D dim5 7] [Eb aug5 maj7] [F dim5 dim7] [G maj7] [Ab maj7] [B min dim7])
```

Scale-tone chords are used primarily to determine all possible analysis of a chord. According to the context (e.g. the neighboring chords in the sequence), the right analysis will be chosen (Cf. the analysis of jazz chord sequences). The following method yields all possible tonalities in which a chord may belong, according to the existing scale classes:

```
(Chord new fromString: 'C maj') possibleTonalities          ->
OrderedCollection (
{V of F HungarianMinor}          {VI of E HungarianMinor}          {IV of G MelodicMinor}
{V of F MelodicMinor}   {I of C Major}                     {IV of G Major}
{V of F Major}                 {V of F HarmonicMinor}            {VI of E HarmonicMinor})
```

4. Temporal objects

An other important aspect of MusES is the representation of time. Our representation of time is based on a simple inheritance scheme. A root class (`TemporalObject`) defines a `startTime` and a `duration`, expressed in fractions of a beat. Notions having a temporal extent are defined as subclasses of `TemporalObject`: `OctaveDependentNote`, `MusicalSilence` and `OctaveDependentChord` (and not `Chord`). Note that although a lot of musical notions have temporal extent, not all of them do!. For instance, we thought it was important to represent pitch-classes independently of any representation of time, because all their properties are indeed time-independent. Hence, the classes representing pitch-classes (seen above) are not subclasses of `TemporalObject`. On the contrary, octave-dependent notes are represented as having an explicit temporal extension, because no interesting time-independent property was found.

Temporal sequences of notes are called *melodies*. A melody holds a list of octave-dependent notes. We distinguish between monophonic polyphonic melodies, because the latter are strongly connected with the 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 (Walker & al. 1992). 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, particular knowledge bases, incorporating lots of knowledge about musical notation. For instance, the problem of knowing which way to draw beams (up or down), how and when to group eighth or sixteenth notes together, how to split notes whose durations exceed certain amounts of time (syncopations), where to position notes and so forth, are problems that do require lot of musical knowledge in order to be solved (Cf. e.g. Ross 87). 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/cut/paste`, `fileIn/out`, 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 tutorial system about musical notation.

5. Extensions

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

5.1. 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 tonalities for each chord in a sequence, when possible. Previous approaches to this problem where mostly based on a formal theory of the underlying domain. For example, Steedman (84) uses context-dependent grammar rules to model 12-bar blues, that capture all "legal" distortions from the original 12-bar blues sequence. However his model is not directly implementable, and yields solutions only for well formed chord sequences. On the contrary, our approach 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, Turnarounds, 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 MusES with classes to represent chord sequences and objects used for the analysis (the well-known shapes, the analysis themselves, etc.). The reasoning is represented by rule bases expressed in NéOpus, a first-order forward-chaining inference engine integrated with Smalltalk-80 (Cf. Pachet (1995)). 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 & Perrot (1994)).

5.2. 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 (Ebcioglu (1991), Chen (1991)) is the overuse 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 showed already very promising results in terms of efficiency (Pachet & Roy 1994), compared to previous attempts by Chen (1991) and Ballesta (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 & Ganascia (1994) has been developed, and is used in conjunction with a representation of musical actions or PACTs (Cf. Pachet (1991), Ramalho & Pachet (1994)). The idea is to model the complete sequence of processes involved in improvisation, from the beginning (parsing of the chord sequence, using the chord sequence analyzer mentioned above) up to the actual generation of notes. The generation of PACTs uses the case-based model of memory in conjunction with an algorithm that generates PACTs according to the musician's experience, as well as general knowledge about musical actions.

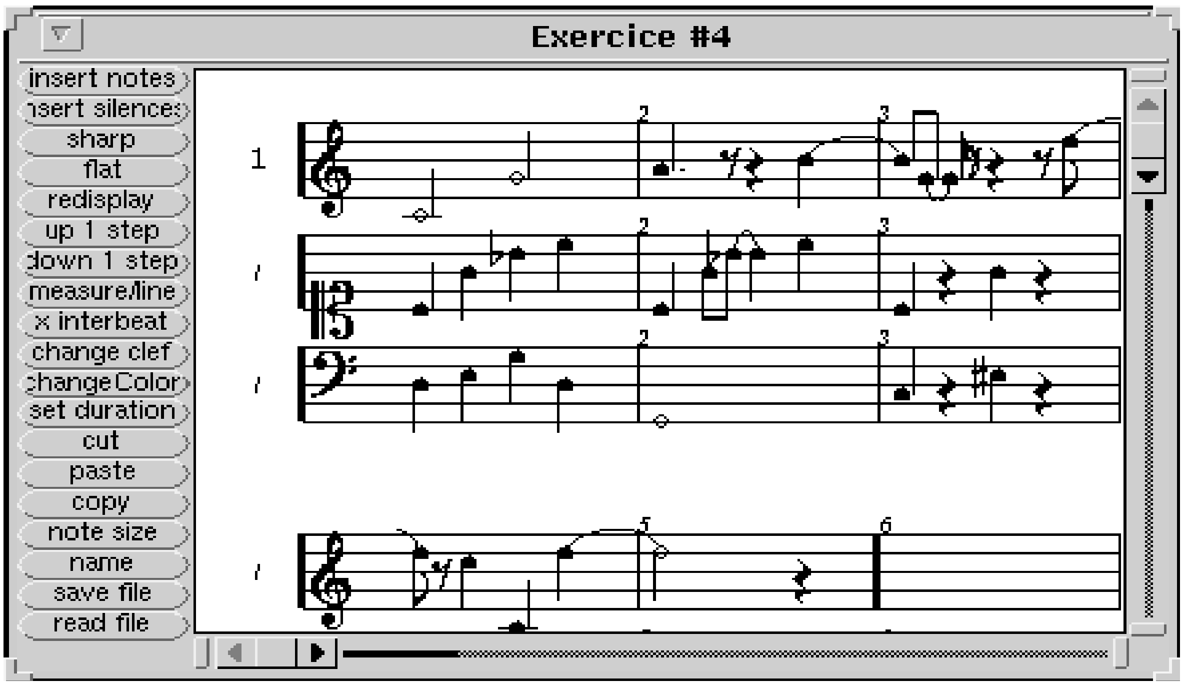

Figure 1. The polyphonic score editor.

6. Discussion, future works

6.1. Why could not I reuse MODE classes ?

Reusing the MODE system seemed a good idea, given the range of problems MODE addresses, the fact that it is written in Smalltalk, a language that many consider as the most reusable of all. However, things did not turn out to be so easy. Almost no classes from MODE could be reused, and we would like to give here a few arguments that may explain why.

- Tonal harmony versus "enharmonic" harmony. Enharmonic spelling is considered in MusES as a central issue, as it supports all analysis reasoning. It cannot be made as an extension of any representation that would not take it into account from the very beginning.
- Analysis versus synthesis. As we said, MODE is oriented towards synthesis (of sounds, music, structures), whereas MusES is more intended to support the construction of reasoning systems. This has unexpected practical effects: reification is not done in the same spirit. For instance, we need in MusES to have intervals represented as both objects **and** operations, and therefore have to represent the corresponding relations with pitch-classes, and octave-dependent notes. This makes the whole music-magnitude classes of MODE unadapted, since these objects are at the bottom of the hierarchy, and support all the system. By comparison, we could imagine how much of the Smalltalk environment would have to be changed if collections were represented by chains of pointers rather than by arrays (as it is the case).
- Representations of time. Our representation of time is also different from the representation of MODE. MODE introduces the notion of *EventList*, as a list of association startTime/musical event. This has the advantage of genericity since the musical events can be any object that define a small set of necessary methods (including event lists themselves). However, it has the disadvantage that musical events do not "know" their `startTime`, either directly (by an instance variable) or indirectly (since they do not have any reference to the `eventList` that hold them). Our representation of time, based on inheritance, solves this problem, and it has the major advantage of being simple to implement. It suffers from the other drawbacks, though. 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".

6.2. Conclusion, future works

We described MusES, a knowledge base that represents concepts of basic harmony and their most current operations. We gave an overview of three systems built on top of MusES, that use MusES structures in conjunction with various inference mechanisms. Future work includes: 1) the connection of the graphical editors with MusiTex, 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 a simple kind), as well as Schenkerian analysis; and 4) using MusES and its extensions as a tutorial system.

7. References

Ballesta, J. (1994). Contraintes et objets: clefs de voûte d'un outil d'aide à la composition. *Journées d'informatique Musicale*, Bordeaux, march 1994.

Cointe, P. Rodet, X. (1991) Formes: Composition and Scheduling of Process. In *The Well-Tempered Object: Musical Applications of Object-Oriented Software Technology* , S. T. Pope, ed. MIT Press.

Ebcioglu, K. (1992). An Expert System for Harmonizing Chorales in the Style of J.-S. Bach, In M. Balaban, K. Ebicioglu & O. Laske (Ed.), *Understanding Music with AI: Perspectives on Music Cognition*, The AAAI Press, California.

Holland, S. (1994). Learning About Harmony Space: An Overview. M. Smith, A. Smaill & G. Wiggins (Ed.), *Music Education: an Artificial Intelligence Perspective*, Springer-Verlag, London.

Pachet, F. (1991) A meta-level architecture for analysing jazz chord sequences. *International Conference on Computer Music*, pp. 266-269, Montréal, Canada.

Pachet, F. (1991b) Representing Knowledge Used by Jazz Musicians. *International Conference on Computer Music*, pp. 285-288, Montréal, Canada.

Pachet, F. (1994). An object-oriented representation of pitch-classes, intervals, scales and chords. *Journées d'informatique Musicale*, Bordeaux, march 1994.

Pachet, F. (1994b) A Refined Framework for Representing Knowledge Based on Simulation. *Colloque Langages et modèles à objets*, Grenoble, octobre 1994, to be published.

Pachet, F. (1995) On the embeddability of production systems in object-oriented languages. *Journal of Object-Oriented Programming* , Dec. 1995. To be published.

Pachet, F. & Perrot, J.-F. (1994). Rule Firing with Metarules. *Software Engineering and Knowledge Engineering* . SEKE '94, Jurmala, Lettonie. Knowledge System Institute Ed. pp. 322-329, 21-23 june 1994.

Pachet, F. & Roy, P. (1994). Mixing constraints and objects: a case study in automatic harmonization. *Proc. of TOOLS Europe 95*. Versailles, June 1995.

Pope, S. (1991). Introduction to MODE: The Musical Object Development Environment. In *The Well-Tempered Object: Musical Applications of Object-Oriented Software Technology* , S. T. Pope, ed. MIT Press.

Ramalho, G., Pachet, F. (1994). From real book to real jazz performance. *International Conference on Music Perception and Cognition*, Lièges, Belgium, july 1994.

Ramalho, G., Ganascia, J.-G. (1994). Simulating Creativity in Jazz Performance. *Proc. of 12th AAAI conf.* Seattle, aug. 1994.

Ross, T. (1987). Teach Yourself: The Art of Music Engraving and Processing. Hansen House, 1987.

Scaletti, C. (1987). Kyma: An Object-oriented Language for Music Composition. in *Proceedings of the International Computer Music Conference*. International Computer Music Association, San Francisco.

Steedman, M.J. (1984). A Generative Grammar for Jazz Chord Sequences. *Music Perception*, Fall 1984, Vol. 2, N° 1, pp. 52-77.

Walker, W., Hebel, K., Martirano, S., Scaletti, C. (1992). ImprovisationBuilder: improvisation as conversation, *Proc. of ICMC* , 1992.

Winograd, T. (1993). Linguistics and the Computer Analysis of Tonal Harmony. In *Machines Models of Music*, Edited by S. M. Schwanauer and D.A. Levitt, MIT Press.