Musical Materials and Algorithmic Composition

Carlos Eduardo Mello

This article dicusses a definition of *musical material* and its role in computer assisted algorithmic composition systems. It describes the implementation of this concept within the MuM project, developed by the author as a programming framework for the creation of automated composition systems.

keywords: algorithmic composition, material

Materiais Musicais e Composição Algoritmica

Este artigo discute uma definição de *material musical* e seu papel em sistemas de composição algorítmica por computador. Descreve a implementação deste conceito no projeto MuM, desenvolvido pelo autor como um framework de programação para a criação de sistemas de composição automatizados.

palavras-chave: composição algorítmica, material

Introduction

Algorithmic composition is by no means a new area of study. Over the last few decades many composers and researchers have designed and implemented various solutions for employing computational processes as a driving force in the genesis of music. These efforts range from experiments in building simple melodic lines (Todd 1989), through reconstructing the style of Bach chorales (Ebcioglu 1988; Hörnel and Menzel 1998), to mapping sophisticated statistical methods to sound parameters (Doornbusch, 2002). A great deal of work must be employed in the implementation of the code necessary to put in practice this formalization of the compositional process. Many composition systems result from years of experimenting and programming. Often much of the labor involves the creation of the computational environment on which to build an algorithm; an environment which accounts for the needs of musical concepts and structures.

Since the early experiments of Hiller and Isaacson (Ariza 2005), there has always been a great deal of interest in a discrete concept of sonic events, in which the organization of notes assumes a prominent place. David Cope's EMI (1992) and CUE (1997) systems, and Bruce Jacob's *Variations* (1996) are only a few examples. This type of approach to automated music composition, which models music ideas in terms of notes, motifs, phrases, and so on, is the main concern of our proposed algorithmic composition framework. As will be seen later, musical abstractions in musical terms can provide an intuitive and flexible environment for developing composition algorithms.

Context

Many algorithmic composition systems documented to date, aim at emulating a composer's methodology. The development of these systems involve modeling data structures which describe musical concepts, as well as defining procedures for manipulating these data structures. While the structures and methods thus developed are usually fit for the task at hand, they can rarely be scaled

for use in other approaches. Yet, it would be hard to deny the existence of a common ground in music making: music is built by organizing sounds in time. This in itself implies that certain common questions need to be addressed. When does a given sound start? How long does it last? What does it sound like? What other sounds occur before/during/after it? Such questions, and many others, when asked in the context of a computer program, reveal the general necessity of representing the basic characteristics of music events in computationally viable ways. Particularly, within systems which are based in a relatively traditional approach to music composition, the basic problems are similar, even though stylistic solutions may differ substantially. Thus, it seems logical to assume that many composers interested in algorithmic methods would benefit from a musical framework which provides a common language for the manipulation of music data.

Cristopher Ariza's *Open Design for Computer-Aided Algorithmic Music Composition* (2005), is a recent attempt to provide a low level environment, which allows the encoding of various models of automated music composition. During the course of his research, Ariza makes a comprehensive survey of known computer aided algorithmic compositional systems (CAACs). After examining various examples of what he calls "idiom constraints", he concludes that no single system can be used to produce every possible style of music. As a consequence of these stylistic constraints in most algorithmic composition systems, composers who want to exploit automated processes in composition often need to develop their own systems. This usually involves programming a computer and dealing with many time consuming details of handling data. Even the most simple model of music information involves a considerable amount of code in order to build a reasonably functional system.

The **MuM** project for algorithmic composition, presented in this article, proposes the use of an object oriented language, aided by a framework of music related classes, as a means of facilitating the construction of new compositional systems. The programming language chosen for this project was C++, a widespread and standardized language, with a full array of OOP features. By using the language's facilities for overloading functions and operators and defining new types, the "dialect" thus created aids the composer in implementing useful algorithms in a more intuitive way, and without writing as much code. The MuM framework embodies some built in limitations, resulting from design choices. First of all, it assumes a discrete concept of musical events. In other words, music is made by assembling notes with defined lengths and pitches. Furthermore, the pitch representation employed assumes the use of tempered tuning. Efforts have been directed towards building a model of music representation which emphasizes traditional manipulation of musical materials, such as motivic development and related procedures. While the framework only offers a

limited set of basic transformations to music data, it provides an easy way of building more complex procedures by abstracting much of the house keeping involved in data manipulation.

Musical Materials

One of the key concepts for the model presented here is the idea of *musical material*. This term is frequently used in discussions about traditional music composition and analysis. It is usually associated with many different elements of music, including a single note, an interval, a rythmic cell, a melodic motif, a phrase, a chord progression, or an entire section of a piece. In this context, just about any musical idea or part of it, can serve as raw material, based on which a composer can weave a complete musical work.

From the description above it is possible to extract what is probably the most important characteristic of our concept of musical material: flexibility. If any combination of notes can be treated as a musical material, than any transformations applied to musical materials produces more musical materials. In fact, history shows that many enduring pieces of music composition present structures which reveal the workings of many transformations upon a relatively small collection of basic materials. The greater components constructed from these smaller materials, often serve as units for building larger structural elements, thus acting as new materials. Conversely, large materials such as a section or phrase can be dissected into shorter passages, which in turn are transformed and regrouped to construct yet new phrases. The relationships found within such pieces of music reinforce the evidence that a discrete yet scalable unit of music representation may be of assistance in the development of composition algorithms. Such a powerful unit naturally lends itself to be implemented through an object oriented model.

Ariza's model, mentioned earlier, does employ object oriented methodology to construct the basic elements of its composition environment. He applies the term "musical material" to describe a collection of abstractions used to represent musical data in various levels. For Ariza, however, "musical material" is a general name under which he groups various categories of data representation, in contrast with "musical procedure", which encompasses the processes used to manipulate this data. The actual classes arrived at by his model define each of these individual categories, which include separate representations for pitch, rhythm, events and instruments.

The model proposed here, on the other hand, differs fundamentally from Ariza's design, in that the idea of *musical material* is abstracted into one single class of objects. The flexibility referred to

earlier is the defining characteristic of this class. Its individual instances act as black boxes, taking any musical elements thrown at them and allowing a number of fundamental transformations to be applied to its internal data. It is precisely this abstraction which allows an intuitive use of the programming language in order to build compositional systems.

If we now set out to construct such a framework, the first question that comes to mind is: how to represent music? Well, obviously this is not a question which can be answered easily. By limiting the scope of the question to the context mentioned above, things become more manageable, although still not easy. We are attempting to adapt traditional approaches to music composition to the realm of algorithmic processes. The idea is to develop a system for facilitating the manipulation of note-based information, which is frequently organized in various patterns such as intervals, melodies, chords, phrases, etc. Whatever compositional style may be involved, some undeniable properties of musical information must be included in our definitions. For example, we must acknowledge the relations between octaves and pitch classes and the fact that similar intervalic patterns are usually perceived as melodically equivalent. This idea would naturally lead to a model which included various forms of pitch transposition. Another very important aspect of human perception of music is our ability to identify groups of notes which have similar rhythmic proportions but different absolute durations. This would call for time scaling capabilities in our framework. We would probably need to devise ways of accommodating music representations which are focused on harmonic as well as polyphonic (contrapuntal) writing. Thus we should also include provision for storing and manipulating independent voice content.

The MuM framework deals with these and other issues by abstracting them into the Musical Material class. This class acts as an open container. From the inside, it has a specific structure which organizes music events in the form of notes and places them in ordered lists. This internal structure however, although documented, is completely transparent to the user, who is expected to deal with it as a black box. So from the outside, all the user needs to know is that he/she can put anything inside the material: a single note, a melody, a chord, etc. Once inside a musical material the data can be modified in various ways. It can be transposed, rescaled, segmented, distributed into various voices, copied to other materials, duplicated, etc. This use of the language's features to encapsulate all this functionality into one flexible object makes using this type of object as simple as any built in data type.

For example, to start using a Musical Material we only need to declare it as we would declare any other variable. To copy its entire content, we use the familiar assignment operator. In order to add two materials together, so that one starts where the other one ends, we simply use the addtion

operator. If, instead, we need to mix the two materials together, we multiply the two variables.

```
MuMaterial mat1, mat2, phrase, melody, chords;
// ...
mat2 = mat1;
phrase = mat1 + mat2;
phrase = melody * chords;
```

Fig.1 - *Musical Material: basic operations*

Every variable used in the previous example (**Fig. 1**) is of the same type - an instance of the MuMaterial class. Though this may seem trivial, from a programming perspective, it should be remembered that each operation illustrated above involves several internal instructions which are handled transparently by the framework, whatever the complexity of the materials involved. That is, objects are constructed and memory allocated and released as necessary; data is copied and modified; several calculations are made regarding start time adjustments for each note, voice assignment, etc. Wether these objects contain a single note or an eight-part counterpoint texture, the composer can use the same simple syntax to combine them.

Data input from outside a program can be handled by loading a text score file in Csound format. The method bellow automatically loads notes from the score into various voices of the 'tune' material, according to instrument assignments in the score (**Fig. 2**).

```
tune.LoadScore("melody.sco");
```

Fig. 2 - *Material input from text file (Csound)*

Output can be done in one of two formats: Csound or MIDI. Both types of output are achieved by simply calling a single method from any material and providing a file name. This facilitates debugging while building the composition system. We can take instant 'snapshots' of any material in any step of the program (**Fig. 3**).

```
composition.Score("myComp.sco");
phrase.SMF("myComp.mid");
```

Fig. 3 - *Output formats: Csound and MIDI*

Beyond these basic operators, the MuM framework provides a number of transformation methods which perform various tasks to modify an initial data set, generating new musical materials. The following example shows some of these methods at work:

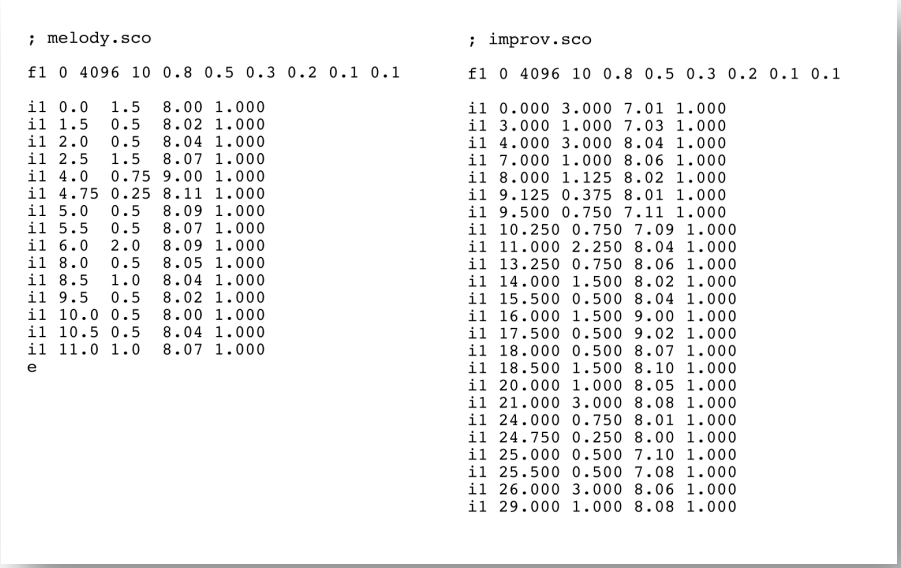
```
#include "MuMaterial.h"

int main(void)
{
 MuMaterial melody, segments, temp, improv;          // 1
 MuInit();                                           // 2
 melody.LoadScore("melody.sco");                     // 3
 segments = melody.Segments(6);                      // 4

 while( improv.Dur() < 30 )                          // 5
 {
  temp = segments.GetVoice( Between( 0, 2 ) );       // 6
  temp.Transpose( Between( -12, 12 ) );              // 7
  temp.Fit( Between( 2, 4 ) );                       // 8
  improv = improv + temp;                            // 9
 }
 improv.SetFunctionTables(melody.FunctionTables()); // 10
 improv.Score("improv.sco");                        // 11

 return 0;
}
```

Figure 4 shows a brief example of how simple commands can produce quite a bit of variety in the manipulation of musical data. First, a few musical materials are declared (1) and the framework is initialized (2). Then, a melodic line is loaded into one of them from a Csound score file (3). Following, the melody is divided into 6 segments of equal length and each one is placed in a different voice of the 'segments' material (4). The code then enters a loop which will produce 30 seconds of improvised melodic lines, employing the segments extracted from the original melody (5). The segments are chosen at random (6), then transposed (7) and rescaled (8). Finally they are appended to the 'improv' material (9), which is later otput to a new score file (11). Although, this short code excerpt would hardly qualify as a complete algorithmic composition program, it does give some hints as to the possibilities of the framework. **Figure 5** shows a sample run of the program with the contents of input ("melody.sco") and output ("improv.sco") files. Besides input from score files, the MuM framework provides classes for specifying note parameters directly and for handling errors. It also supplies the composer with a much wider range of basic transformations then space allows to demonstrate in the present article.

Conclusion

Although the model of music representation presented here does not yeld itself directly to any particularly new approach to the algorithmic composition problem, it should be able to contribute in a useful way by simplifying future investigations in this field. The idea of a general purpose musical

material, in which music data can assume diverse shapes and functions, provides a powerful way of conveying information within the system and reduces the need for dealing with low level maintnance. Future developments should include constructing more complex trasnformation methods and connecting these to higher level processes for controling transformation choices and generating large scale music structure.



```
; melody.sco

f1 0 4096 10 0.8 0.5 0.3 0.2 0.1 0.1

i1 0.0   1.5   8.00 1.000
i1 1.5   0.5   8.02 1.000
i1 2.0   0.5   8.04 1.000
i1 2.5   1.5   8.07 1.000
i1 4.0   0.75  9.00 1.000
i1 4.75  0.25  8.11 1.000
i1 5.0   0.5   8.09 1.000
i1 5.5   0.5   8.07 1.000
i1 6.0   2.0   8.09 1.000
i1 8.0   0.5   8.05 1.000
i1 8.5   1.0   8.04 1.000
i1 9.5   0.5   8.02 1.000
i1 10.0  0.5   8.00 1.000
i1 10.5  0.5   8.04 1.000
i1 11.0  1.0   8.07 1.000
e
```

```
; improv.sco

f1 0 4096 10 0.8 0.5 0.3 0.2 0.1 0.1

i1 0.000  3.000  7.01 1.000
i1 3.000  1.000  7.03 1.000
i1 4.000  3.000  8.04 1.000
i1 7.000  1.000  8.06 1.000
i1 8.000  1.125  8.02 1.000
i1 9.125  0.375  8.01 1.000
i1 9.500  0.750  7.11 1.000
i1 10.250 0.750  7.09 1.000
i1 11.000 2.250  8.04 1.000
i1 13.250 0.750  8.06 1.000
i1 14.000 1.500  8.02 1.000
i1 15.500 0.500  8.04 1.000
i1 16.000 1.500  9.00 1.000
i1 17.500 0.500  9.02 1.000
i1 18.000 0.500  8.07 1.000
i1 18.500 1.500  8.10 1.000
i1 20.000 1.000  8.05 1.000
i1 21.000 3.000  8.08 1.000
i1 24.000 0.750  8.01 1.000
i1 24.750 0.250  8.00 1.000
i1 25.000 0.500  7.10 1.000
i1 25.500 0.500  7.08 1.000
i1 26.000 3.000  8.06 1.000
i1 29.000 1.000  8.08 1.000
```

Fig. 5 - *Sample run: input and output*

References:

ARIZA, Christopher. An Open Design for Computer-Aided Algorithmic Music Composition: athenaCL. Boca Ratton, FL: Dissertation.com, 2005.

COPE, David. Computer Modeling of Musical Intelligence in EMI. Computer Music Journal, Vol. 16, No. 2. MIT Press, Summer 1992.

COPE, David. The Composer's Underscoring Environment: CUE. Computer Music Journal, Vol. 21, No. 3, pp. 20-37. MIT Press, Summer 1997.

DOORNBUSCH, Paul. Composers views on mapping in agorithmic composition. Organised Sound, 7(2), --. 145-156. Cambridge University Press, 2002.

EBCIOGLU, Kemal. An Expert System for Harmonizing Four-part Chorales. In Computer Music Journal, Vol. 12 No. 3. MIT Press, Fall 1988.

JACOB, Bruce. Algorithmic composition as a model of creativity. Organised Sound, 1(3), --. 157-165. Cambridge University Press, 1996.

HÖRNEL, D. and MENZEL, W. Learning Musical Structure and Style with Neural Networks. In Computer Music Journal, 22:4, pp. 44-62. MIT Press, winter 1998.

TODD, Peter. A connectionist Approach to Algorithmic Composition. Computer Music Journal, Vol. 13 No. 4. MIT Press, Winter 1989.