Hierarchical Music Representation for Composition and Analysis

Alan Smaill, Geraint Wiggins, Mitch Harris [*]

Abstract

In this paper, we present intermediate results of continuing research into the utility of generalised hierarchical structures for the representation of musical information. We build on an *abstract data type* presented in [Wiggins *et al* 89], using *constituents*, which are structurally significant groupings of musical events. We suggest that a division into such groupings can be musically meaningful, and that it can be more flexible than similar approaches. We demonstrate our representation system at work in both analysis and composition, with output from computer programs. We conclude that it is possible and useful to represent music in a way independent of the particular style, tonal system, *etc*, of the music itself.

1 Introduction

This paper addresses the problem of representation of music on computers. In it, we present first steps towards a *general musical representation system*: that is to say, a representation system which is not based in any particular style, tonal system, tradition, or even application. To demonstrate that our work is indeed this general, we will present our system operating in two different kinds of computer language (ML, a functional programming language, and Prolog, a logic programming language), with examples of both computer aided composition and analysis, the latter being in two different tonal systems, and two different representations.

The point of all this is to foreshadow an eventual situation where a researcher in computer music could use any computer program with his or her chosen representation system, limited only by the suitability of the program for computation over the data represented[1]. This said, claims to be closely approaching such a situation at this stage are out of the question. Before a truly generalised representation for music can be attained, a much deeper understanding of the issues involved is required. The work presented in this paper is part of an attempt to move towards such an understanding.

[*]Department of Artificial Intelligence, University of Edinburgh, 80 South Bridge, Edinburgh EH1 1HN, Scotland; Email:{A.Smaill,geraint,M.Harris}@uk.ac.edinburgh

[1]This limitation is quite reasonable. For example, we would not generally expect to generate a Bach-style harmonisation for a melody expressed in a microtonal scale.

The structure of the presentation is as follows. First, we briefly recapitulate the ideas presented in our earlier paper, [Wiggins *et al* 89], since this forms the foundation for the new work presented here. We then introduce the idea of the *constituent* with an example of automated composition. Next, we illustrate automated analysis using the system, with a fuller description of the brief example of the earlier paper (the analysis of Debussy's "Syrinx" for solo flute, under an analysis method due to Ruwet [Ruwet 72, Nattiez 75]). This is presented with two different representations of the same music, which are analysed with identical results by the same program, illustrating the power of the abstract data type. Finally, this program is applied to another piece, Part I (Largo) of Ives' "Three Quarter-Tone Pieces for Two Pianos", demonstrating that, given a technique which is applicable (*viz* Ruwet's style of analysis), our representation allows software developed in one tonal system to be directly used in another.

2 Representing Musical Events

2.1 The Abstract Data Type

In [Wiggins *et al* 89], we presented a basic system for the representation of music *events* – that is to say combinations of pitch, onset time, duration and timbre – which we claimed was adequate for many compositional and analytic purposes. We did not then, and we do not now, claim that such a system is adequate for the representation of all aspects of music; for example, it does not easily allow specification of envelopes. However we do claim that it is adequate for study of the kind of structural information widely agreed (see *eg* [Balaban 88]) to be inherent in music.

Given this basic set of information types, we showed that the notation of music might be *abstracted* in such a way that the detail of the notation was invisible to any applications operating over it. This division between representation and application was achieved by the use of interface functions: simple, low level functions which must be defined by the user in terms of his or her representation. These functions either return output in terms of standard computer language types (in particular, Boolean, or true/false), or in terms of the representation they define. As long as the set of functions is complete, and the user's use of them exclusive, the question of how precisely the representation is implemented never arises, because it is irrelevant for the purposes of analysis.

2.2 Example: An Implementation of the Abstract Data Type

Let us take for an example one of the representations used for Debussy's "Syrinx" later in this paper. This, like all representations fulfilling our specification, has notational forms for pitch, pitch interval, onset time and duration. Each of the types so denoted is ordered in the obvious way, with a distinguished symbol denoting zero. Basic functions are then typed equivalents of arithmetic comparison functions (*eg* $\leq$, $\neq$), operations (*eg* $+$, $-$), and access functions (*eg* given an event identifier, get the pitch associated with that event).

In this example, pitch is a triple of $\langle$note name,accidental,octave$\rangle$ (where accidental is one of { bb b = # x }), intervals are integer number of semitones, and onset times and durations are conventional note names, ties being denoted by +. As an abbreviation, integer multipliers may precede note values, like this: `2*minim`. The first line of "Syrinx", scored as in Figure 1, is then represented by the (Prolog) clauses in Figure 2.

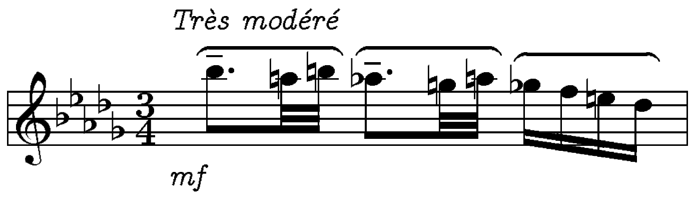

Figure 1: The first bar of Debussy's score of "Syrinx"

```
event(e000,[b,b,1],0,dotted_quaver).
event(e001,[a,=,1],dotted_quaver,demisemiquaver).
event(e002,[b,=,1],dotted_quaver+demisemiquaver,demisemiquaver).
event(e003,[a,b,1],crotchet,dotted_quaver).
event(e004,[g,=,1],dotted_crotchet+semiquaver,demisemiquaver).
event(e005,[a,=,1],dotted_crotchet+dotted_semiquaver,demisemiquaver).
event(e006,[g,b,1],minim,semiquaver).
event(e007,[f,=,1],minim+semiquaver,semiquaver).
event(e008,[e,=,1],minim+quaver,semiquaver).
event(e009,[d,b,1],minim+dotted_quaver,semiquaver).
event(e010,[b,b,1],dotted_minim,dotted_quaver).
event(e011,[c,=,2],dotted_minim+dotted_quaver,demisemiquaver).
event(e012,[c,b,2],dotted_minim+(dotted_quaver+demisemiquaver),demisemiquaver).
event(e013,[b,b,1],dotted_minim+crotchet,minim).
```

Figure 2: The first bar of "Syrinx" in our first example representation

Suppose that, for the purposes of an analysis, we wish to find if the first and third notes in this phrase are equal in pitch. This could be expressed in a functional programming language as

eqpitch(getpitch(e000), getpitch(e002))

or in a logic programming language as

getpitch(e000, X), getpitch(e002, Y), X eqpitch Y.

The important thing to notice in this example is that we do not have to refer to the elements of the datatype itself: events are accessed through their identifiers, which are universal. Thus, it makes no difference what is the form of our exact representation, as long as the necessary interface functions are defined. The whole issue then rests upon the existence of a small atomic set of such functions; in the earlier paper, we argued that such a set does exist. This work is intended to further justify and refine that claim.

2.3 Constituents

In [Wiggins *et al* 89] we gave a small example of our representation system at work, by partially reproducing an analysis due to [Nattiez 75] using a technique of [Ruwet 72]. This technique is particularly well suited to our purposes, since it relies on the "bottom up" technique of finding similarities between different phrases, rather than the more conventional "top down" approaches. We will reproduce this example in Section 4, and use it to justify the existence of a particular kind of constituent, the *stream*. The example will show how the constituent system allows us to represent structural information without compromising the need also to make musical annotations.

We will start with the score of "Syrinx" notated as in Figure 2; it would be boring to reproduce the whole here. The example is constructed in the Prolog programming language. The events constituting the score are asserted into the Prolog database, along with a specification of the fact that they constitute a particular piece. We can do this with a constituent, in Figure 3.

```
constituent( co1,
             collection( 0, 34*dotted_minim+crotchet),
             syrinx,
             [e000,e001,...e181,e182] ).
```

Figure 3: Collection constituent naming "Syrinx"

The meanings of the four fields of the constituent are as follows. The first is a unique identifier for the constituent, whose form is the choice of the implementer. The fourth is a set of identifiers, naming the events which make up the constituent; these events are then the *particles* of the constituent. The third field can be thought of as a "musical type" – that is, a label which, in some sense, represents the role of the constituent. In this case, we (rather arbitrarily) represent the fact that this group of events makes up the piece "Syrinx", by giving that name as the musical type. The usage of this field will become clearer in the example below. The second field, the *structural type*, specifies the nature of the constituent. In this case, the constituent is to be viewed as an entity with a start time and an end time, but no further visible structure. The *collection*, then, is the least structured, and therefore the least interesting, kind of constituent.

2.4 Stream Constituents

Now, it so happens that we can do something more useful than this. Our data is a piece for solo flute. Therefore (at least naïvely) it is monophonic. Therefore, the set of events making up the "Syrinx" constituent is strictly ordered.

In the example, below, it will become clear that we can use this fact to improve the efficiency of our analysis. Also, it seems that many groupings of interest in music are likely to exhibit this property of strict ordering – most melodies, for example, are monophonic, or can be viewed as strictly ordered sequences of chords. It seems likely that it will be useful, then, to admit the existence of a a class of constituents enjoying this property. This we will call a *stream*. A stream has a start time and a duration, and, as a first approximation, we will suggest that its particles are contiguous[2]. Given, then, that the particles of the "Syrinx" constituent are strictly ordered, we can rewrite it as shown in Figure 4.

```
constituent( co1,
             stream( 0, 34*dotted_minim+crotchet),
             syrinx,
             [e000,e001,...e181,e182] ).
```

Figure 4: Stream constituent naming "Syrinx"

So now, when we call our analysis procedure (called with the identifier of a constituent – co1 in this case) we can use the structural type to determine whether or not to call a version of the program optimised for monophonic lines. This is a typical simple example of the use of structural types.

Note also, that, so long as we assign each constituent a unique identifier, there is no reason why we should not allow the simultaneous existence of multiple forms. This means that we can represent multiple "views" of the same data using constituents with different "musical types", which is an important requirement of a musical representation system.

3 Hierarchies of Constituents and Composition

We now describe how the basic representation and the constituent mechanism together can be used to allow flexible access to musical structures and manipulation of these structures in a way that might be useful to a composer.

3.1 Implementing the Basic Representation

The basic representation has been implemented in the functional typed programming language ML [Wikström 87]; this indicates the generality of our approach. Standard ML is equipped with a module system, which allows us to reflect abstract data typing together with different implementations in the programming language itself. Unlike in Prolog, we can insist that the abstract data typing is respected, and have the language check that this is indeed the case using the compile-time type checking mechanism of the language.

To give a brief idea of how this works, consider the representation of pitch. The form of the corresponding abstract data type is given by a declaration of a *signature* saying what types and functions are involved. For pitch, this is shown in Figure 5. Thus there

[2]This means that explicit rest events must be inserted into any gaps.

```
signature PITCHSIG =
    sig
        type pitch
        type interval
        val inval: pitch * pitch -> interval
        val less_inval: interval * interval -> bool
        val add_inval: pitch * interval -> pitch
    end;
```

Figure 5: ML signature for the basic pitch representation

are types for pitches and intervals, a function to calculate intervals between pitches, a comparison function for interval size, and a function to calculate a pitch from a given pitch and an interval.

We may then write programs that assume the existence of types and functions as in the signature. For example, a function to take two pitches and return pitches at twice the interval apart can be written as

```
fun double ( p1:PITCHSIG.pitch, p2:PITCHSIG.pitch ) =
       ( p1 , add_inval( p1 , add_inval ( p1 , inval ( p1 , p2 ) ) ) )
```

and the language will type check this expression. (Here `PITCHSIG.pitch` is the type of `pitch` as supplied from the signature `PITCHSIG` above.)

Subsequently the signature can be implemented in some concrete data type. For ex-

```
structure pitch1 =
    struct
        datatype octave =  octave of int
        datatype degree = degree of int
        type pitch = octave*degree
        datatype interval = Interval of int
        fun inval((octave(n1),degree(m1)),(octave(n2),degree(m2))) =
                                        Interval(12*(n2-n1)+(m2-m1))
        fun less_inval(Interval(x),Interval(y)) = x<y
        fun add_inval ( (octave n, degree m) , Interval i ) =
                            let val p = 12*n + m + i in
                                (octave (p div 12) , degree (p mod 12))
                            end;
    end;
```

Figure 6: ML implementation of the basic pitch representation

ample, Figure 6 shows a concrete datatype that implements the pitch structure described above .

The details are not important, but note that pitches are represented by pairs of integers, tagged as representing octave and degree within the octave respectively, and intervals as the number of degrees between pitches. Again, a claim in the language that this structure contains objects with the right properties to implement the abstract pitch datatype is verified by the type checker. The language then allows us to take this datatype and plug it into programs written over the abstract type, such as the `double` function above to obtain an executable program.

3.2 Implementing Constituents

We claim that the basic representation contains in principle enough information to allow a wide range of musical tasks to be performed. However, it is vital that the user can approach the musical material in ways that make musical sense at a higher level, and that the user has liberty in choosing how the material can best be structured for the purpose at hand. Our intention is that the constituent mechanism allows the user just this freedom.

To illustrate a possible compositional use, consider the sorts of operations that underly Ligeti's use of micropolyphony, for example in the Chamber Concerto, or the second string quartet. These proceed by the progressive deformation of small motifs, for example stretching out the pitch contours of the material, squashing the rhythms used, perhaps cutting off the material completely whenever it passes over some pitch sill, throwing the material from instrument to instrument, and so on. The listener is not aware of the detail of these processes which can appear as generating a knotted texture, rather than some more conventional musical development.

We suggest that something of the spirit of this technique can be captured by combining functions that perform a number of operations of the form described, perhaps including some random element. The ability to experiment easily with different combinations of transformation would be helpful to the composer – linking the output of such a program up to some sound generation device is not difficult.

For example, we suppose a motif given in the form of a stream constituent with some number of events.

```
constituent( c0 , stream( 0,t1 ), motif, [e1,e2,e3,e4,e5] ).
```

We can in fact use virtually the same syntax for constituents in the ML version as for Prolog.

Operations that can easily be written include

- Dilation of the interval structure between events: given a dilation parameter, produce a constituent whenever the intervals of pitch are enlarged. Using nearest available pitches in a non-continuous pitch system breaks up the uniformity of this procedure; we can also use versions where the dilation is not uniform.
- Similar operations for rhythm use virtually identical code.

- Replacement of events by sub-constituents: we can use the motif as a framework in which events are replaced by something more complex, such as chords (depending on the event), other motifs, and so on.
- Blanking out of some material: it is easy to simply omit according to some criterion some small moments of the material, and so break up what might become too uniform a structure.
- Instrumentation can be dealt with by using distinct constituents for each instrument.
- Operations such as inversion in pitch and time can be regarded as special cases of the dilation operation.

Combining such operations allows the composer to manipulate musical material using a small number of musically meaningful operations, and yet generate structures of considerable complexity.

4 Structural Analysis of Music: An Example in Two Notations

4.1 The Analysis Procedure

Let us proceed, then, to the first analysis example. [Ruwet 72]'s procedure for musical analysis is attractive because of its simplicity. It relies on the notion of *similarity* between different parts of a piece of music. Some similarities are as simple as *identity* (between, say, statements and restatements of a motif) and some are sufficiently complicated that extensive knowledge of music is necessary to encode them. For the purposes of experiment, we have focussed on the simpler similarities, since the harder ones are sufficiently complex as to cause significant drain on resources – if we used them, the time required for experimentation (especially in the polyphonic example given later) would become impractical.

Given these various kinds of similarity, then, Ruwet gives a simple algorithm for using them for musical analysis. We have adapted this algorithm slightly, to make it run more quickly over a large search space. Our algorithm runs as follows.

- First, the music is scanned for phrases which are repeats of earlier phrases. When a repeat is found, that phrase is given a special status. We will call its first occurrence a *motif*, and view the repeats as *derivations* under the *identity* similarity. As each of these special status phrases is found, it is removed from the search space of the rest of the program.
- After the repeats have been removed, the piece is scanned for phrases which are similar to the motif under the other known similarities. These, too, are given special status as derivations under the relevant similarity; they are removed from the search space of the next pass of the program.

- This procedure is repeated until no further repeated phrases can be found, whereupon it terminates, usually leaving some notes unassigned to any significant phrase.

This version of the algorithm is different from that of Ruwet in that it requires a phrase to be repeated before it is considered under the non-identity similarities. Because of the inclusion of the transposition similarities in the search, this constraint reduces the search space considerably, while (at least for this data) maintaining the interesting results.

4.2 The Similarities

The similarities used in our analysis are as follows:

- Identity – True if a phrase is identical with another.
- Longer Identity – True if a phrase is almost identical with an existing motif – specifically, identical except for a longer first note.
- Transpose – True if a phrase is a transposition of a constant number of semitones (*ie* not a modal transposition) of an existing motif.
- Loose Transpose – True if a phrase is a transposition of a constant number of semitones (*ie* not a modal transposition) of an existing motif, regardless of the equality of note durations.

It might be argued that these similarities are rather *ad hoc*; indeed, it is generally necessary to use similarities in this kind of analysis which are suited to the style of writing used by the composer. However, note that the program is modular over the set of similarities, which can be regarded as further data which can easily be updated.

4.3 Representing the Output

This analysis procedure, then, like most such procedures, gives us output in terms of groupings of the set of notes which were its input. We can clearly represent these groups with collection constituents. However, the groups output in this analysis are all strictly ordered contiguous sequences of notes. Therefore, we can represent our output using stream constituents; for example, the first motif found by the analysis program is exactly the bar shown in Figures 1 and 2, which is identical with the third bar. The part of the output of the program representing the two phrases and their relationship is shown in Figure 7. The complete set of output constituents, all of which are streams, is shown in the appendix.

Note how the "musical type" field shows the special status of the phrases, and identifies the motif with a name, which is referred to in the derived phrase.

As well as asserting these new constituent structures into the Prolog database, the program writes a commentary as it proceeds. This is shown in the appendix. Note that

```
constituent(co1,stream(0,dotted_minim),motif(mtf2),
            [e000,e001,e002,e003,e004,e005,e006,e007,e008,e009]).
constituent(co3,stream(3*dotted_minim,dotted_minim),derived(identity,mtf2),
            [e014,e015,e016,e017,e018,e019,e020,e021,e022,e023]).
```

Figure 7: Part of the output of the analysis program

the output is in terms of identifiers only, except for the inclusion of interval specifications for the transposition similarities. These intervals are output in the particular implementation of the representation – thus, -12 is one octave down.

4.4 The Same Analysis from a Different Representation

To demonstrate the power of the representation system, let us now take a different implementation of the data type. This time, we will represent pitches by integer numbers of quartertones, with Middle C being defined as 128. Intervals will also be integer numbers of quartertones. Onset times and durations are represented by rational fractions of demisemiquavers.

The output commentary for this analysis turns out to be identical with that of the previous section, except for interval values, which are this time in quartertones, according to the representation.

4.5 Displaying the Results

We have written a program which displays the results of our analysis in the style of Nattiez's presentation (FigureA.4); the correspondence between our results and Nattiez's (in [Nattiez 75, p332]) is clearly visible. As we use a different set of similarities from Nattiez, our analysis is not identical, but the analyses agree to a large extent.

5 Extending the Analysis to Another Tonal System

5.1 The Data

There are two main issues to consider when attempting to apply our program to other, harder sets of data. The first is the problem of the choice of appropriate similarities, mentioned before; the second is the problem of polyphony[3].

As for the former, Ruwet's technique is only as suitable to a given style of music as the similarity specifications it uses. We suggest that the similarities we have defined are sufficiently simple and general to apply to some extent in many cases.

[3]Here we use the word in its weak sense: that is to say, meaning non-monophonic.

The polyphony issue is equally easily dealt with, at the expense of greatly increased non-determinism in the analysis program. Now, instead of relying on the strict ordering of the events in the "Syrinx" constituent, we must search the collection constituent specifying our new data as we go along, for streams. Search for each "next event" is then a matter of calculating the next onset time (*ie* `addtd( gettime( e1 ), getdur( e1 ))`) where `e1` is the current event. The process can be optimised by sorting the data in advance, so that it is at least partially ordered.

The main point of interest of this second analysis is that the piece, "Largo" from the "Three Quarter-Tone Pieces for Two Pianos", is written in a different tonal system from "Syrinx". Nevertheless, using an appropriate representation (and the second one from the "Syrinx" example in Subsection 4.4 is ideal), we can use the Ruwet technique to analyse it – clearly, similarities based on identity and fixed-interval transpositions will not be affected by the change of tonal system.

5.2 Possible Variations in the Analysis

With the introduction of polyphony comes another potential aspect of interest in the analysis. In "Syrinx" we were always looking for monophonic streams, because there was no other kind of structure available in the music. Now, we have more options: for example, some of the similarities obvious from the score are between related *chord* sequences, rather than *note* sequences.

The version of the analysis program presented here does not understand similarities between chords – and it turns out that under the simple similarities used here, useful results can still be obtained (see below). This is unsurprising: there is an obvious equivalence between a sequence of chords and a set of monophonic phrases, each with the same rhythm. However, in musical terms, there are certainly cases where it would be more correct to express similarities between chord sequences than between the phrases constituting them. (We must also point out that there are many cases where unfelicitous grouping of notes into chords may obscure a previously visible similarity.)

The modular nature of the analysis program suggests that we could easily design similarity clauses over sets of notes as well as individuals (indeed, first steps in this direction have been taken). All we would need, then would be a representation admitting streams of chords – the stream still being the basic unit over which the analysis works. The idea of a chord can clearly be captured in our structural type system as a collection of events; however, it seems likely that we will want to define a particular structural type, orthogonal to streams, which captures the notion of a chord more precisely. This will be a subject for future consideration.

Given similarities defined between collections of simultaneous notes we might suggest an iterative process where we first pick out truly monophonic phrases. Subsequently remaining chords could be found, represented as constituents, and then a further pass of the analysis program could find similarities based on these. Again, this will be a subject for future work.

5.3 The Results

The "Largo" analysis is less interesting than that of "Syrinx", on account of the optimisation based on identity checking in Section 4.1; it turns out that (in general) Debussy was particularly liberal with repeats in his music. Thus, the existing program fails to find all the similarities in "Largo" because they are not based on repeated motifs. This, however, is an implementation detail.

Even so, the analysis program does arrive at a limited structural result which is meaningful in terms of the musical sectioning of the piece, and succeeds in identifying musically important themes and motifs. Because the music is polyphonic, Ruwet's representation style requires adaptation to yield worthwhile results.

6 Related Work and Conclusions

A variety of formal models of musical structures have been proposed. Our time representation allows time to be given in topological terms, so setting it apart from [Diener 88] and [Chemillier & Timis 88], who have a more restricted notion. Often these formal models describe musical structures in grammatical terms. Our work is orthogonal to this approach – a grammar could be specified in terms of the constituent structure, and a piece of music parsed by determining automatically whether such a constituent structure could be imposed upon the underlying events. Our interest in describing the internal structure, in pitch and time, of the musical events, sets our work apart from most grammatical approaches where the notion of event is taken as a primitive with no internal structure.

The results of the experiments presented here suggest that there is some utility in proposing a representation based on our abstract data type. We have proposed the existence of (at least) two special kinds of constituent, whose structural properties, regardless of musical interpretation, can be relevant in analysis. Our intention is to continue to attempt to identify such structures, so that automatic procedures may be specified in terms of them. We have demonstrated the power of our representation, and thus given weight to our suggestion that such a representation can be in some sense universal.

References

[Balaban 88] M. Balaban. A music-workstation based on multiple hierarchical views of music. In C. Lischka and J. Fritsch, editors, *Proceedings of the 14th International Computer Music Conference*, pages 56–65, 1988.

[Chemillier & Timis 88] M. Chemillier and D. Timis. Towards a theory of formal musical languages. In C. Lischka and J. Fritsch, editors, *Proceedings of the 14th International Computer Music Conference*, pages 175–83, 1988.

[Diener 88] G. Diener. Ttrees: an active data structure for computer music. In C. Lischka and J. Fritsch, editors, *Proceedings of the 14th International Computer Music Conference*, pages 184–88, 1988.

[Nattiez 75] J.-J. Nattiez. *Fondements d'une sémiologie de la musique.* Union Générale d'Editions, Paris, 1975.

[Ruwet 72] N. Ruwet. *Langage, musique, poésie.* Editions du Seuil, Paris, 1972.

[Wiggins *et al* 89] G. Wiggins, M. Harris, and A. Smaill. Representing music for analysis and composition. In C. Lischka, editor, *Proceedings of the 2nd IJCAI AI/Music Workshop*, 1989.

[Wikström 87] A. Wikström. *Functional Programming using Standard ML.* Prentice Hall, London, 1987.

A Appendix: Program Input and Output

A.1 "Syrinx" Input Constituent

```
constituent(co000,stream(0,824/1),syrinx,
          [e000,e001,e002,e003,e004,e005,e006,e007,e008,e009,e010,e011,
           e012,e013,e014,e015,e016,e017,e018,e019,e020,e021,e022,e023,
           e024,e025,e026,e027,e901,e902,e903,e904,e905,e906,e907,e908,
           e909,e028,e029,e030,e031,e910,e911,e912,e913,e914,e915,e033,
           e034,e035,e036,e037,e038,e039,e040,e041,e042,e043,e044,e045,
           e046,e047,e048,e049,e050,e051,e052,e053,e054,e055,e056,e057,
           e058,e059,e060,e061,e062,e063,e064,e065,e066,e067,e068,e069,
           e070,e071,e072,e073,e074,e075,e076,e077,e920,e921,e922,e079,
           e080,e081,e082,e083,e084,e085,e086,e087,e088,e089,e090,e925,
           e928,e926,e927,e092,e093,e094,e930,e931,e932,e933,e935,e938,
           e936,e937,e940,e942,e943,e944,e945,e948,e946,e947,e099,e100,
           e101,e102,e950,e952,e953,e954,e104,e105,e750,e752,e753,e754,
           e107,e108,e109,e110,e955,e956,e957,e958,e112,e113,e960,e961,
           e962,e963,e965,e966,e967,e968,e969,e116,e970,e971,e972,e118,
           e119,e120,e121,e800,e801,e802,e803,e804,e805,e806,e807,e808,
           e809,e810,e123,e124,e125,e126,e127,e128,e129,e130,e131,e500,
           e501,e502,e503,e504,e505,e506,e507,e508,e509,e510,e511,e133,
           e600,e601,e602,e603,e604,e605,e606,e607,e608,e609,e610,e611,
           e135,e136,e137,e138,e139,e140,e141,e142,e143,e144,e145,e146,
           e147,e148,e149,e150,e151,e152,e153,e154,e155,e156,e157,e158,
           e159,e160,e161,e162,e163,e164,e820,e821,e822,e823,e824,e825,
           e166,e167,e168,e169,e830,e831,e832,e833,e834,e835,e171,e172,
           e173,e174,e840,e841,e842,e843,e176,e856,e857,e178,e179,e860,
           e861,e862,e863,e181,e182]).
```

A.2 "Syrinx" Output Constituents

```
constituent(co1,stream(0/1,24/1),motif(mtf2),
    [e000,e001,e002,e003,e004,e005,e006,e007,e008,e009]).
constituent(co3,stream(48/1,24/1),derived(identity,mtf2),
    [e014,e015,e016,e017,e018,e019,e020,e021,e022,e023]).
constituent(co4,stream(192/1,24/1),derived(transpose(-24),mtf2),
    [e039,e040,e041,e042,e043,e044,e045,e046,e047,e048]).
constituent(co5,stream(216/1,18/1),derived(transpose(-24),mtf2),
    [e049,e050,e051,e052,e053,e054,e055]).
constituent(co6,stream(592/1,32/1),derived(longer_identity,mtf2),
    [e140,e141,e142,e143,e144,e145,e146,e147,e148,e149]).
constituent(co7,stream(640/1,32/1),derived(longer_identity,mtf2),
    [e155,e156,e157,e158,e159,e160,e161,e162,e163,e164]).
constituent(co8,stream(672/1,16/1),derived(loose_transpose(-24),mtf2),
    [e820,e821,e822,e823,e824,e825,e166,e167,e168,e169]).
constituent(co9,stream(688/1,24/1),derived(loose_transpose(-24),mtf2),
    [e830,e831,e832,e833,e834,e835,e171,e172,e173,e174]).
constituent(co10,stream(72/1,8/1),motif(mtf11),
    [e024,e025,e026]).
constituent(co12,stream(96/1,8/1),derived(identity,mtf11),
    [e028,e029,e030]).
constituent(co13,stream(312/1,8/1),motif(mtf14),
    [e925,e928,e926,e927]).
constituent(co15,stream(336/1,8/1),derived(identity,mtf14),
    [e930,e931,e932,e933]).
constituent(co16,stream(344/1,8/1),derived(identity,mtf14),
    [e935,e938,e936,e937]).
constituent(co17,stream(360/1,16/3),derived(transpose(-2),mtf14),
    [e945,e948,e946]).
constituent(co18,stream(408/1,16/3),derived(transpose(-2),mtf14),
    [e750,e752,e753]).
constituent(co19,stream(1096/3,104/3),motif(mtf20),
    [e947,e099,e100,e101,e102,e950,e952,e953,e954,e104]).
constituent(co21,stream(1240/3,104/3),derived(identity,mtf20),
    [e754,e107,e108,e109,e110,e955,e956,e957,e958,e112]).
constituent(co22,stream(460/1,4/1),motif(mtf23),
    [e960,e961,e962]).
constituent(co24,stream(1528/3,4/1),derived(identity,mtf23),
    [e803,e804,e805]).
constituent(co25,stream(536/1,8/3),motif(mtf26),
    [e500,e501,e502,e503]).
constituent(co27,stream(1616/3,8/3),derived(identity,mtf26),
    [e504,e505,e506,e507]).
constituent(co28,stream(1624/3,8/3),derived(identity,mtf26),
    [e508,e509,e510,e511]).
constituent(co29,stream(552/1,8/3),motif(mtf30),
    [e600,e601,e602,e603]).
constituent(co31,stream(1664/3,8/3),derived(identity,mtf30),
    [e604,e605,e606,e607]).
constituent(co32,stream(1672/3,8/3),derived(identity,mtf30),
    [e608,e609,e610,e611]).

constituent(c033,stream(0/1,824/1),ruwet(syrinx),
           [co1,e010,e011,e012,e013,co3,co10,e027,e901,e902,e903,e904,
            e905,e906,e907,e908,e909,co12,e031,e910,e911,e912,e913,e914,
            e915,e033,e034,e035,e036,e037,e038,co4,co5,e056,e057,e058,
            e059,e060,e061,e062,e063,e064,e065,e066,e067,e068,e069,e070,
            e071,e072,e073,e074,e075,e076,e077,e920,e921,e922,e079,e080,
             e081,e082,e083,e084,e085,e086,e087,e088,e089,e090,co13,e092,
             e093,e094,co15,co16,e940,e942,e943,e944,co17,c019,e105,co18,
             co21,e113,c022,e963,e965,e966,e967,e968,e969,e116,e970,e971,
             e972,e118,e119,e120,e121,e800,e801,e802,co24,e806,e807,e808,
             e809,e810,e123,e124,e125,e126,e127,e128,e129,e130,e131,co25,
             co27,co28,e133,co29,co31,co32,e135,e136,e137,e138,e139,co6,
             e150,e151,e152,e153,e154,co7,co8,co9,e840,e841,e842,e843,
             e176,e856,e857,e178,e179,e860,e861,e862,e863,e181,e182]).
```

A.3 Commentary output for first example representation

```
Performing analysis in the style of Ruwet of syrinx
 Found repeated phrase mtf2 at e000 and e014
 Searching for other occurrences of mtf2
  Found all occurrences of mtf2
 Searching for phrases related to mtf2
  Found phrase similar to mtf2 under transpose(-12) at e000 and e039
  Found phrase similar to mtf2 under transpose(-12) at e000 and e049
  Found phrase similar to mtf2 under longer_identity at e000 and e140
  Found phrase similar to mtf2 under longer_identity at e000 and e155
  Found phrase similar to mtf2 under loose_transpose(-12) at e000 and e820
  Found phrase similar to mtf2 under loose_transpose(-12) at e000 and e830
  Found all phrases similar to mtf2 under known transformations
 Found repeated phrase mtf11 at e024 and e028
 Searching for other occurrences of mtf11
  Found all occurrences of mtf11
 Searching for phrases related to mtf11
  Found all phrases similar to mtf11 under known transformations
 Found repeated phrase mtf14 at e925 and e930
 Searching for other occurrences of mtf14
  Found repeated phrase mtf14 at e925 and e935
  Found all occurrences of mtf14
 Searching for phrases related to mtf14
  Found phrase similar to mtf14 under transpose(-1) at e925 and e945
  Found phrase similar to mtf14 under transpose(-1) at e925 and e750
  Found all phrases similar to mtf14 under known transformations
 Found repeated phrase mtf20 at e947 and e754
 Searching for other occurrences of mtf20
  Found all occurrences of mtf20
 Searching for phrases related to mtf20
  Found all phrases similar to mtf20 under known transformations
 Found repeated phrase mtf23 at e960 and e803
 Searching for other occurrences of mtf23
  Found all occurrences of mtf23
 Searching for phrases related to mtf23
  Found all phrases similar to mtf23 under known transformations
 Found repeated phrase mtf26 at e500 and e504
 Searching for other occurrences of mtf26
  Found repeated phrase mtf26 at e500 and e508
  Found all occurrences of mtf26
 Searching for phrases related to mtf26
  Found all phrases similar to mtf26 under known transformations
Scan complete.d phrase mtf30 at e600 and e604
Ruwet analysis finished
```

A.4 The display of our results

```
Analysis of syrinx in the style of Ruwet


mtf2   e010-e013
mtf2.1           mtf11   e027-e909
                 mtf11.1           e031-e038
mtf2.2
mtf2.2                                       e056-e090 mtf14   e092-e094
                                                       mtf14.1
                                                       mtf14.1           e940-e944
                                                       mtf14.3                     mtf20
                                                       mtf14.3
                                                                                   mtf20.1 e105-e113 mtf23   e963-e802
                                                                                                     mtf23.1           e806-e131 mtf26
                                                                                                                                 mtf26.1
                                                                                                                                 mtf26.1 e133 mtf30
                                                                                                                                              mtf30.1
                                                                                                                                              mtf30.1 e135-e139
mtf2.4                                                                                                                                                          e150-e154
mtf2.4
mtf2.5
mtf2.5                                                                                                                                                                    e840-e182


Transformations used:
1. identity
2. transpose(-24)
3. transpose(-2)
4. longer_identity
5. loose_transpose(-24)
```