LRN2CRE8

LEARNING TO CREATE

PROJECT NUMBER: 610859

SMALL OR MEDIUM-SCALE FOCUSED RESEARCH PROJECT

ICT - FUTURE AND EMERGING TECHNOLOGIES (FET)

Deliverable 1.2
Encoding language

AAU Version 1.0 (Final)

Authors / **Contributors**:

David Meredith (AAU)

Executive Summary: This report provides a description of the current version of a prototype implementation of a geometric music encoding language called MEL (which stands for Music Encoding Language). The motivation behind MEL and its roots in psychological coding theories and information theory are explained. Then the fundamental concepts of MEL are introduced. Of these concepts, perhaps the most important is that of a *masked space*, which extends Deutsch and Feroe's [4] concept of a pitch alphabet and generalizes it to the representation of metrical and rhythmic structure. This allows chords, scales, repeated rhythms and metric structures to be encoded parsimoniously. Another important feature of MEL is the *product* function which returns the cartesian product of any number of vector sum sets given to it as arguments and applies these to a set of notes. In this way, large complex musical objects can be generated from small note patterns translated by the product of two or more vector collections. Several examples of MEL encodings of musical excerpts are provided in order to illustrate the features of the language.

Dissemination Level		
PU	Public	X
PP	Restricted to other programme participants (including the Commission Services)	-
RE	Restricted to a group specified by the Consortium (including the Commission Services)	-
CO	Confidential, only for members of the Consortium (including the Commission Services)	-

Table of Contents

1 OVERVIEW

In fulfilment of deliverable D1.2, “Encoding language”, of the *Learning to Create* project (Lrn2Cre8), this document provides a description of a prototype implementation of the current version of a geometric music encoding language called MEL (which stands for Music Encoding Language), an early version of which was described by Meredith [10]. In § 2, the motivation behind the development of MEL will be explained and its background in research on psychological coding theories and information theory will be reviewed. The fundamental concepts in MEL will then be introduced in § 3. Finally, in § 4, the current version (MEL23) of a prototype implementation of MEL (in Java) will be described. A more detailed report on the MEL language and its associated encoding and decoding algorithms will be provided in deliverable D1.3, “Reports on encoding language”, which will be completed in month 36 of the Lrn2Cre8 project.

2 MOTIVATION AND BACKGROUND

Music analysis is about finding the best ways of understanding musical works. In other words, it is concerned with finding the most successful *perceptual organizations* that are consistent with some given musical “surface”. The success of a particular analysis (way of understanding, perceptual organization) can be evaluated either subjectively (as is typically done in traditional humanistic music analysis) or objectively by determining the extent to which a given analysis helps with carrying out some objectively evaluable task (e.g., error detection, composer recognition). The musical surface is typically a representation of either a notated score or a particular performance. Of the two, a score usually permits a higher number of consistent perceptual organizations, since the micro-structure of a performance usually reflects the particular perceptual organization (i.e., interpretation) of the performer.

Most theories of perceptual organization have been based on one of two principles: the *likelihood* principle of preferring the most *probable* interpretations (originally due to Helmholtz [24]); and the *minimum* [23] or *simplicity* [3] principle of preferring the *simplest* interpretations. Statistical approaches to musical structure analysis, such as ones based on Markov models, Shannon entropy or Bayesian inference (e.g., [14,22]) have applied the likelihood principle, whereas theories in the tradition of Gestalt psychology (e.g., [4]) apply the minimum principle. Indeed, as van der Helm and Leeuwenberg [23, p. 153] point out, the fundamental principle of Gestalt psychology, Koffka’s [5] law of *Prägnanz*, which favours the simplest and most stable interpretation, can be seen as an “ancestor” of the minimum principle. The likelihood and minimum principles were considered by psychologists to be in competition until Chater [3], in 1996, drawing on Kolmogorov’s [6] theory of complexity, pointed out that the two principles are mathematically identical.

The MEL language, presented in § 3, is in the tradition of the Gestalt-based coding languages for music proposed by Simon and Sumner [18], Restle [15] and Deutsch and Feroe [4]. It attempts to generalize and extend earlier languages by adopting a geometric approach, along the lines of that proposed by Meredith *et al.* [13]. A passage of polyphonic music is represented in the proposed language as a set of multidimensional points, generated by performing geometric transformations on component patterns. The language introduces the concept of a periodic *mask*, a generalization of Deutsch and Feroe’s [4] notion of a pitch alphabet, that can be applied to any dimension of a geometric representation, allowing for both rhythms and pitch collections to be represented parsimoniously in a uniform way.

In the context of Lrn2Cre8, a language like MEL can be used to encode automatically generated analyses of all the pieces in a corpus representing some target style or genre. These analyses can then themselves be mined for more abstract regularities that apply to more than one of the resulting analyses. The discovered regularities can then be used in a music generation system to improve the coherence and quality of generated music intended to be in the target style or genre.

The earliest coding language for music was proposed in 1968 by Simon and Sumner [18], who recognized that music is multidimensional and aimed to allow for the description of patterns in melody, harmony, rhythm and form. Their language treats each of these dimensions as an independent series of symbols, chosen from alphabets, for which a compact representation can be derived in terms of certain basic element operators, such as SAME and NEXT. Simon and Sumner defined the notion of a

cyclic alphabet and recognized the usual tonal scales and chords as “common” pitch alphabets that are “ordered sets already defined in the culture”.

In his experiments on serial pattern learning, Restle [15] found that subjects are particularly good at identifying “runs” (e.g., (2 3 4 5)) and “trills” (e.g., (5 4 5 4)) and tend to use these to segment a series of symbols. Restle explained this by proposing that runs and trills allow for particularly simple generative descriptions. He presented an “E-I theory” wherein a rule consists of a set E of “events” (roughly equivalent to an alphabet) and a set I of “intervals” (in the musical sense). Runs are then the set of products of E-I rule systems in which I contains only one interval. Restle’s language uses the sequence operators M (for mirror), T (for transposition) and R (for repeat) for producing compact descriptions of sequences. For example, if the numbers 1 to 6 represent a row of 6 lights that can either be on or off, then the sequence (1 2 1 2 2 3 2 3 6 5 6 5 5 4 5 4) can be encoded as M(T(R(T(1)))). Restle represented structures in his language as hierarchical trees and presented an analysis of the theme of Bach’s first Two-part Invention (BWV 772) which describes its tonal and motivic structure.

Deutsch and Feroe’s [4] model is in the tradition of the serial pattern languages proposed by Restle [15], Leeuwenberg [7] and Simon [17]. The language can be used to encode arhythmic monophonic sequences of pitches (i.e., neither polyphony nor rhythm can be encoded). *Structures* are defined to be sequences of the elementary operators ‘same’ (s), ‘next’ (n) and ‘predecessor’ (p), that operate over *alphabets*, which are linearly ordered sets of symbols. Structures are decoupled from alphabets. A *sequence*, $\{S;\alpha\}$, is the application of a structure S to an alphabet, α; and a “sequence of notes” (actually a sequence of pitches) can be generated by applying a sequence to a *reference element*. *Compound sequences* can be built up from sequences by using the sequence operators, ‘prime’ (pr), ‘retrograde’ (ret), ‘inversion’ (inv) and ‘alternation’ (alt). Alphabets can be defined as sequences and ‘stacked’ hierarchically. For example, the C major scale would be defined as a subset of the chromatic scale (denoted by ‘Cr’), using the expression $\mathrm{C} = \{\{(*, 2n^2, n, 3n^2, n); \mathrm{Cr}\}; \mathrm{c}\}$ and the C major triad could be defined relative to the C major scale by the expression $\{\{(*, 2n^2, n^3); \mathrm{C}\}; 1\}$. The MEL language presented in this report extends and generalizes these notions of structures and alphabets by re-casting them in geometric terms.

As mentioned above, in the context of Lrn2Cre8, the motivation for developing a coding language like MEL is to provide a means for expressing analyses of musical works and corpora, that can then form a body of knowledge to be used for generating new works in the styles represented by the analysed pieces. The term *musical analysis* is used here to mean a particular way of understanding certain structural aspects of a *musical object*, where such an object may be any quantity of music, ranging from a motive, chord or even a single note through to a complete work or even an entire corpus of works (cf. [1, p. 1]). In the spirit of Kolmogorov complexity theory [2, 6, 9, 20, 21] and the minimum description length principle [16], a musical analysis is conceived of here as being a compact or compressed encoding of an *in extenso description* of a musical object. An in extenso description is one in which the properties of each atomic component of the object are explicitly specified, without encoding any structural groupings of these components into higher-level constituents, and without encoding any relationships between atomic components (see [18, 19] for a similar use of the term, “in extenso”). Music theorists and analysts often refer to such in extenso descriptions as “musical surfaces” [8, pp. 3,10–11]. The atomic components themselves might be, for example, notes in a score or events in a MIDI file or sample values in a PCM audio file. Their nature thus depends on both the nature of the object being described (e.g., a specification of what to play or a recording of an actual performance) and the level of detail or “granularity” of the description.

In contrast, while a musical analysis is itself a description of a musical object, it will typically differ from an in extenso description by representing the object as being constructed from *sets* of atomic components that form larger-scale constituents, such as motives, phrases, chords, voices and sections (or possibly even collections of non-contiguous events in a piece or collection of pieces). An analysis will also typically encode *relationships* between such constituents (e.g., repetition, transposition, inversion, elaboration, augmentation and diminution).

The *Kolmogorov complexity* of an object is, roughly speaking, the length in bits of the shortest possible program that generates the object as its only output. In the spirit of Kolmogorov complexity, an analysis is conceived of here to be a program that outputs the in extenso description of a musical object that we want to analyse and explain. The precise way in which such a program generates the in extenso surface description of the object constitutes an hypothesis as to how that surface might have come about. Equivalently, we can thus consider an analysis to be a losslessly compressed encoding of an in

extenso description of a musical object. An analysis is thus an *explanation for* or a *way of understanding* certain aspects of the structure of the musical object that it describes.

Suppose X and Y are two constituent sets of atomic components of a musical object (e.g., two sets of notes forming two occurrences of the subject in a fugue) and that the transformation, T, maps X onto Y (e.g., T could be "shift in time by x quarter notes and transpose by y semitones"). If T can be described more parsimoniously than Y, then the part of the musical surface consisting of X and Y (i.e., $X \cup Y$, or the union of the atomic components in X and Y) can be described more parsimoniously by giving an in extenso description of X together with a description of T, than it can by giving in extenso descriptions of both X and Y. In this way, by identifying structural relationships between constituents of a musical object, a musical analysis can convey at least as much information about that object as an in extenso description on the same level of detail, but may manage to do so more parsimoniously. A musical analysis can thus take the form of a compact description or compressed encoding of a musical object. Of course, a musical analysis usually conveys *more* information than an in extenso description on the same level of detail, since it also typically describes groupings of atomic components into larger constituents and structural relationships between these constituents.

Kolmogorov complexity [2, 6, 9, 20, 21] (also known as *algorithmic information theory*) suggests that the *length* of a program, whose output is an in extenso description of an object, can be used as a measure of the complexity of its corresponding explanation for the structure of that object: if we have two programs in the same programming language that generate the same output, then the shorter of the two will typically represent the simpler explanation for that output.

The level of structural detail on which an analysis (encoded as a program) explains the structure of a musical object is determined by the granularity of the in extenso description that it generates. Typically, much of the detailed structure of an object will not be encoded in the in extenso description generated by an analysis and this omitted structure will therefore go unexplained.

The work presented in this report is based on the assumption that the music analyst's goal is to find the *best possible* explanations for the structures of musical objects. We also assume that these "best possible" analyses will provide us with the knowledge that we need to be able to compose new pieces of music that either most successfully "interpolate" between the pieces of an existing style or most successfully extrapolate from an existing style. This raises the question of how we are supposed to decide, given a pair of alternative explanations for the same musical object, which of these explanations is "better". We can only meaningfully claim that one analysis is "better than" another if it allows us to more successfully carry out some objectively evaluable task; and even then, we can only claim that the analysis is superior *for that task*. Such tasks might include, for example, detecting errors, memorizing pieces, identifying composers (or dates of composition, forms or genres), and predicting how incomplete pieces might be completed. It is possible that the best analysis for carrying out one such task might be different from the best analysis for carrying out another. However, the work that forms the focus of this report is founded on the hypothesis that the best possible explanations for a musical object (i.e., the best analyses for *all* objectively evaluable tasks) are those that are represented by that object's shortest possible descriptions—that is, the descriptions of the object whose lengths are equal to the Kolmogorov complexity of that object.

In general, the Kolmogorov complexity of an object is not computable.[1] This means that, if we have some in extenso description of an object and a compressed encoding of that description, then typically we cannot be sure that the compressed encoding is the shortest one possible. We can, of course, often prove that an encoding is *not* the shortest possible, simply by finding a shorter one. However, in the current context, the non-computability of Kolmogorov complexity does not pose a problem, as we will only use the relative lengths (i.e., information content) of analyses to predict which analyses will serve us better for carrying out musicological tasks. That is, in order to be able to evaluate whether we are making progress, we never really need to know if an analysis is the best possible (although, of course, that would be nice), we only need to be able to predict (at least some of the time) whether or not it will be better than another one.

If we adopt the approach outlined above, then the music analyst's goal becomes that of finding the shortest possible "programs" (i.e., encodings, analyses) that generate the most detailed representations of as much music as possible. In other words, our ultimate goal may be considered to be to compress

[1] Actually prefix complexity, K, is upper semicomputable, but it cannot be approximated in general in a practically useful sense [9, p. 216].

as detailed a description as possible of as much music as possible into as short an encoding as possible [11, 12].

The MEL language, described in § 3, has been designed with this ultimate goal in mind. As suggested above, one compresses an in extenso description by discovering and removing *redundancy* in it. One important type of redundancy occurs when part of an object is related to some other part by an effective (i.e., computable) transformation that can be described simply and that can often be used when describing objects of the given type. To take a simple example, in a piece of music, it often happens that a main theme is repeated several times: this means that the union of the events in the piece in occurrences of such a main theme can be described in a losslessly compressed manner by specifying the first occurrence of the theme in an in extenso fashion (e.g., by listing the notes in it) and then listing the starting positions of all the other occurrences of the theme. Themes are often repeated transposed either to other keys or "modally" within the same key (e.g., C–D–E–C transposed to E–F–G–E). If we represent a piece of music as a set of points in pitch–time space, then such (possibly transposed) re-statements of a theme correspond to patterns of points related by translation. Translation within pitch–time space is therefore a type of transformation that is commonly used to create redundancy in music. An encoding language for music in which encodings are as parsimonious as possible should therefore permit us to express that one set of notes in a musical object can be generated by translating some other set of notes in the object within pitch–time space. Modal transposition is a special case of transposition within a pitch alphabet (in the sense of Deutsch and Feroe [4]), suggesting that our encoding language should also allow us to express translations within pitch–time spaces that have been 'filtered' or 'masked' using such alphabets. As will be explained in the next section, such transformations can be expressed succinctly in MEL.

3 MEL: A GEOMETRIC MUSIC ENCODING LANGUAGE

In this section, the fundamental concepts of MEL will be introduced. Then, in the next section, the current implementation of the language, MEL23, will be described. In this section, normal mathematical notation will be used to express the abstract description of the language concepts; in the next section, actual working code segments will be presented (indicated by being typeset in `teletype` font).

3.1 Notes and vectors

In MEL, a passage of music is represented as a set of *notes*. A *note*, n, is an ordered pair, $n = \langle t, p \rangle$, where $t = \mathrm{t}(n)$ is the *onset time* of the note in tatums[2] and $p = \mathrm{p}(n)$ is the note's MIDI note number.[3] A note is therefore a point in *note space* which is the two-dimensional Euclidean integer lattice in which the x co-ordinate represents time in tatums and the y co-ordinate represents pitch in terms of MIDI note number.

In MEL, a *vector*, v, is an ordered 4-tuple, $v = \langle t, p, \mathbf{M}_\mathrm{t}, \mathbf{M}_\mathrm{p} \rangle$, where $t = \mathrm{t}(v)$ is the *time component* of v, $p = \mathrm{p}(v)$ is the *pitch component* of v, $\mathbf{M}_\mathrm{t} = \mathbf{M}_\mathrm{t}(v)$ is the *time mask sequence* of v and $\mathbf{M}_\mathrm{p} = \mathbf{M}_\mathrm{p}(v)$ is the *pitch mask sequence* of v. $\mathrm{t}(v)$ and $\mathrm{p}(v)$ are integers. $\mathbf{M}_\mathrm{t}(v)$ and $\mathbf{M}_\mathrm{p}(v)$ are *mask sequences*.

3.2 Masks and mask sequences

A *mask sequence*, $\mathbf{M}$, is an ordered set of *masks*, $\mathbf{M} = \langle m_1, m_2, \ldots, m_k \rangle$. A *mask*, m, is an ordered pair, $m = \langle o, s \rangle$, where $o = \mathrm{o}(m)$ is an integer called the *offset* of the mask and $s = \mathrm{s}(m)$ is an ordered set of positive integers called the *structure* of the mask. Each integer in a mask structure is called an *interval*. If m is a mask and i is an integer, then the function $\mathrm{m}(m, i)$ (see Figure 1) returns the *masked*

[2] A *tatum* is the longest duration that evenly divides every event onset and duration in a musical object.

[3] In future versions of the language, it is intended that other properties of notes will be accommodated, such as duration and voice.

```
m(m, i)
1   if i = nil return nil
2   j ← o(m), k ← 0
3   if i ≥ o(m)
4       while j < i
5           k ← k + 1
6           j ← j + s(m)[(k − 1) mod |s(m)|]
7   else
8       while j > i
9           k ← k − 1
10          j ← j − s(m)[k mod |s(m)|]
11  if j = i return k
12  return nil
```

Figure 1: The function $\mathrm{m}(m, i)$, where m is a mask and i is an integer or nil.

```
u(m, i)
1   if i = nil return nil
2   j ← o(m), k ← 0
3   if i ≥ 0
4       while k < i
5           k ← k + 1
6           j ← j + s(m)[(k − 1) mod |s(m)|]
7   else
8       while k > i
9           k ← k − 1
10          j ← j − s(m)[k mod |s(m)|]
11  return j
```

Figure 2: The function $\mathrm{u}(m, i)$, where m is a mask and i is an integer or nil.

value of i for the mask m; and the function $\mathrm{u}(m, i)$ (see Figure 2) returns the *unmasked value* of i for the mask m.

Figure 3 illustrates how a mask is used to map a subset of the integers onto the complete set of integers. In the upper part of Figure 3, the mask $\langle 3, \langle 2, 2, 1, 2, 2, 2, 1\rangle\rangle$ is applied. The mask defines a periodic repeated pattern of intervals on the number line such that successive elements in the pattern are mapped onto successive integers. For example, the mask offset, 3, is mapped onto 0. The next integer that does not map onto nil is 5, which therefore maps onto 1, and so on. This particular mask, $\langle 3, \langle 2, 2, 1, 2, 2, 2, 1\rangle\rangle$, can be used to represent the E♭ major scale; and the structure of this mask, $\langle 2, 2, 1, 2, 2, 2, 1\rangle$, can be used to represent the class of all major scales. Note that, the fact that each mask has an offset, which serves as a "starting point", means that, when a mask is used to represent a scale, it provides both the mode and the "finalis" or tonic—it does not just represent a pitch class set. Thus the descending form of the A melodic minor scale and the C major scale contain the same set of pitch classes, $\{0, 2, 4, 5, 7, 9, 11\}$. However, the mask representing the descending A minor melodic scale would be $\langle 9, \langle 2, 1, 2, 2, 1, 2, 2\rangle\rangle$, whereas the mask for the C major scale is $\langle 0, \langle 2, 2, 1, 2, 2, 2, 1\rangle\rangle$.

Figure 3 also illustrates that the output of one mask can be given as input to another. Thus we can take the range of the mask $\langle 3, \langle 2, 2, 1, 2, 2, 2, 1\rangle\rangle$ and apply the function in Figure 1 to these values with the

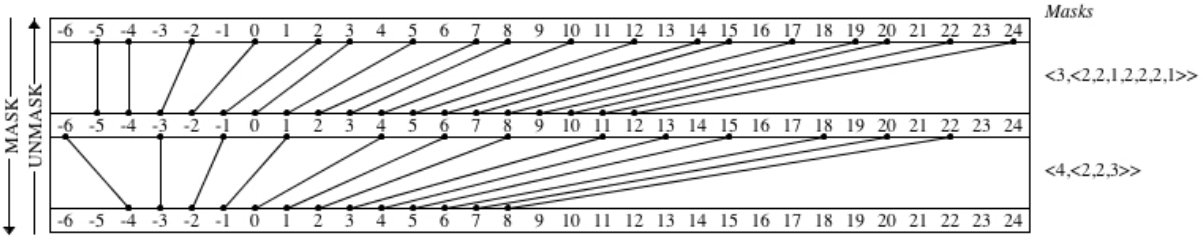


Figure 3: Applying the mask sequence, $\langle\langle 3, \langle 2, 2, 1, 2, 2, 2, 1\rangle\rangle, \langle 4, \langle 2, 2, 3\rangle\rangle\rangle$.

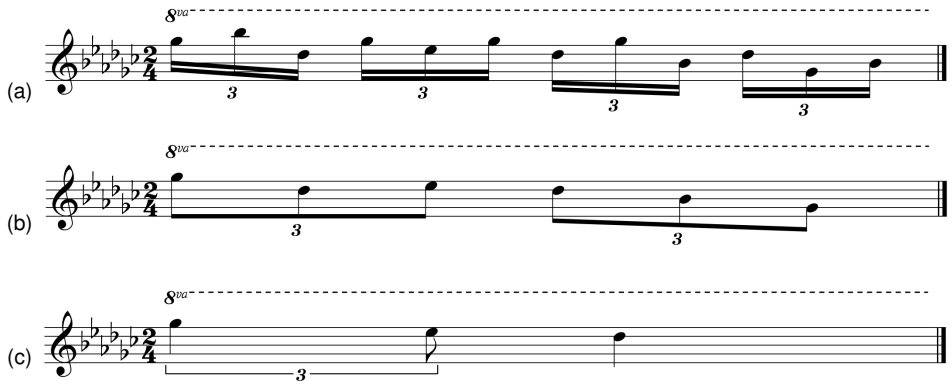


Figure 4: (a) The right-hand part of the first bar of Chopin's *Étude*, Op. 10, No. 5. (b) A plausible rhythmic reduction of (a). (c) A plausible rhythmic reduction of (b).

```
m(M, i)
1   k ← i, j ← 0
2   while j < |M|
3       k ← m(M[j], k)
4       j ← j + 1
5   return k
```

Figure 5: The function $\mathrm{m}(\mathbf{M}, i)$ where $\mathbf{M}$ is a mask sequence and i is an integer.

mask $\langle 4, \langle 2, 2, 3\rangle\rangle$ to give the result shown in the lower part of Figure 3. In this particular case, the mask $\langle 4, \langle 2, 2, 3\rangle\rangle$ is used to represent a dominant triad in any seven-note scale. Applying this to the output of the mask $\langle 3, \langle 2, 2, 1, 2, 2, 2, 1\rangle\rangle$ therefore gives a representation of the dominant triad in E♭ major—i.e., the B♭ major triad. If the topmost number line in Figure 3 represents MIDI note number, then MIDI pitch 10 maps onto 4 in the middle number line, representing the fact that 10 is the fifth scale degree in E♭ major. The number 4 in the middle line is then mapped onto 0 in the lowest number line, representing the fact that this scale degree is now the root of the dominant triad. Thus, the dominant triad in E♭ major can be represented by the mask sequence, $\langle\langle 3, \langle 2, 2, 1, 2, 2, 2, 1\rangle\rangle, \langle 4, \langle 2, 2, 3\rangle\rangle\rangle$. This example demonstrates that a mask sequence can be used to represent the notion of hierarchically related pitch alphabets proposed by Deutsch and Feroe [4]. For example, the mask sequence $\langle\langle 3, \langle 2, 2, 1, 2, 2, 2, 1\rangle\rangle, \langle 4, \langle 2, 2, 3\rangle\rangle\rangle$ corresponds to Deutsch and Feroe's alphabet,

$$\{\{(*, 2n^2, n^3); \{\{(*, 2n^2, n, 3n^2, n); \mathrm{Cr}\}; e♭\}\}; 5\}\ .$$

However, unlike Deutsch and Feroe's pitch alphabets, mask sequences can also be used to represent rhythmic and metric structures. For example, the crotchet metric level in a 4/4 bar in which the tatum is a semiquaver can be represented by the mask sequence $\langle\langle 0, \langle 2\rangle\rangle, \langle 0, \langle 2\rangle\rangle\rangle$. Similarly, the dotted crotchet metric level in a 6/8 bar where the tatum is a semiquaver can be represented by $\langle\langle 0, \langle 2\rangle\rangle, \langle 0, \langle 3\rangle\rangle\rangle$.

Figure 4 (a) shows the right-hand part of the first bar of Chopin's *Étude*, Op. 10, No. 5. Figure 4 (b) and (c) show plausible rhythmic reductions of the surface in Figure 4 (a). If we use the triplet semiquaver as the tatum duration, then the rhythm of Figure 4 (b) can be represented by the mask sequence $\langle\langle 0, \langle 2\rangle\rangle\rangle$; and the rhythm of Figure 4 (c) could be represented by the mask sequence $\langle\langle 0, \langle 2\rangle\rangle, \langle 0, \langle 2, 1, 3\rangle\rangle\rangle$ (or $\langle\langle 0, \langle 4, 2, 6\rangle\rangle\rangle$).

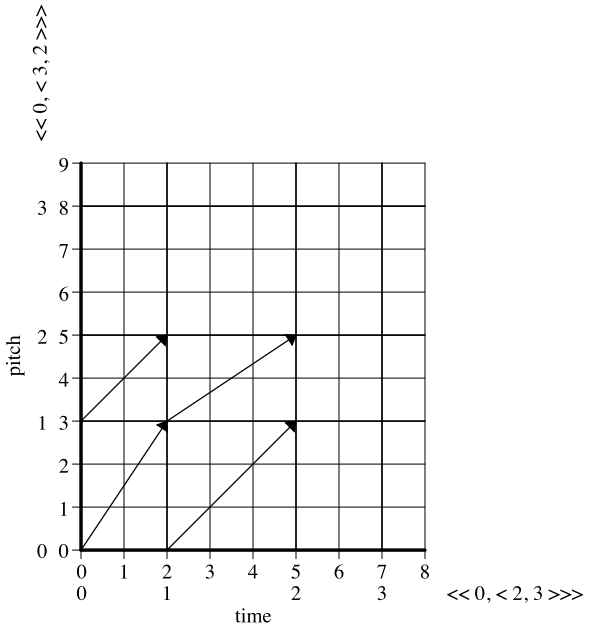
If $\mathbf{M}$ is a mask sequence and i is an integer, then the function $\mathrm{m}(\mathbf{M}, i)$ (see Figure 5) returns the masked value of i for the mask sequence $\mathbf{M}$ (i.e., the result of applying each of the masks in $\mathbf{M}$, in turn, with an initial input value of i). Conversely, the function $\mathrm{u}(\mathbf{M}, i)$ in Figure 6 returns the unmasked value of i for the mask sequence $\mathbf{M}$.

```
u(M, i)
1   k ← i, j ← |M| − 1
2   while j ≥ 0
3       k ← u(M[j], k)
4       j ← j − 1
5   return k
```

Figure 6: The function $\mathrm{u}(\mathbf{M}, i)$ where $\mathbf{M}$ is a mask sequence and i is an integer.

```
T(n, v)
1   x1 ← m(Mt(v), t(n))
2   y1 ← m(Mp(v), p(n))
3   x2 ← x1 + t(v)
4   y2 ← y1 + p(v)
5   t2 ← u(Mt(v), x2)
6   p2 ← u(Mp(v), y2)
7   return ⟨t2, p2⟩
```

Figure 7: The function $\mathrm{T}(n, v)$ where n is a note and v is a vector.



v = <1,1,<<0,<2,3>>>,<<0,<3,2>>>>
T(<0,0>,v) = <2,3> (v is <2,3> in note space)
T(<2,0>,v) = <5,3> (v is <3,3> in note space)
T(<0,3>,v) = <2,5> (v is <2,2> in note space)
T(<2,3>,v) = <5,5> (v is <3,2> in note space)

Figure 8: Equivalent vectors in note space for the vector, $v = \langle 1, 1, \langle\langle 0, \langle 2, 3\rangle\rangle\rangle, \langle\langle 0, \langle 3, 2\rangle\rangle\rangle$.

3.3 Masked spaces

If v is a vector, then $\mathbf{M}_{\mathrm{t}}(v)$ and $\mathbf{M}_{\mathrm{p}}(v)$ together define the *space*, $\mathcal{M}(v) = \langle \mathbf{M}_{\mathrm{t}}(v), \mathbf{M}_{\mathrm{p}}(v) \rangle$, in which v is defined. If a vector, v, is in note space, then the vector can be written as an ordered pair, $\langle \mathrm{t}(v), \mathrm{p}(v) \rangle$, without specifying the time and pitch mask sequences (e.g., $\langle 2, 3 \rangle = \langle 2, 3, \langle\langle 0, \langle 1 \rangle\rangle\rangle, \langle\langle 0, \langle 1 \rangle\rangle\rangle\rangle$). Given a note n_1 and a vector v, then we can *translate* n_1 by v to give a new note, n_2. In order to do this, n_1 must first be mapped onto a point, q_1, in the space $\mathcal{M}(v)$. q_1 is then translated by v in the usual way to another point, q_2, in $\mathcal{M}(v)$. Finally, q_2 is mapped back onto a note, n_2, in note space. If n is a note and v is a vector, then this process of translation can be accomplished using the function in Figure 7.

Note that, if v is a vector in any space other than note space, then there will not, in general, be a unique vector in note space to which v is equivalent. For example, if $v = \langle 1, 1, \langle\langle 0, \langle 2, 3 \rangle\rangle\rangle, \langle\langle 0, \langle 3, 2 \rangle\rangle\rangle$, then Figure 8 shows that there are 4 distinct vectors in note space to which v might be equivalent, depending on the note that is being translated. A consequence of this is that, in general, there is no unique vector in any space that is equivalent to the sum of two or more vectors that are in different spaces. The resultant vector in note space of applying two vectors in succession to a note depends on the note itself. This means that vectors in the sense defined here cannot be added together in the normal way. Instead, if a note is to be translated by the sum of two or more vectors, the vector sum has to be explicitly stated.

3.4 Vector sums

Since the sum of two or more vectors is not necessarily equal to a unique vector in any space, it must be considered a different type of object from a vector. We therefore define a special type of object called a *vector sum* to represent a sum of vectors. A vector sum is an *ordered set* of vectors, since vector addition in the sense defined here is not commutative. If we want to denote the sum of the vectors $v_1, v_2, \ldots v_k$, applied *in that order*, then we simply write $v_1 + v_2 + \ldots + v_k$. If w is the vector sum $v_1 + v_2 + \ldots + v_k$, then $|w| = k$, $w[j] = v_{j+1}$ and $w = \sum_{i=1}^{k} v_i$. If w_1 and w_2 are vector sums such that $w_1 = v_{1,1} + v_{1,2} + \ldots + v_{1,m}$ and $w_2 = v_{2,1} + v_{2,2} + \ldots + v_{2,n}$, then $w_1 + w_2 = v_{1,1} + v_{1,2} + \ldots + v_{1,m} + v_{2,1} + v_{2,2} + \ldots + v_{2,n}$. If v is a vector, then $\mathrm{w}(v)$ returns the vector sum that contains just v. In other words, $\mathrm{w}(v)$ typecasts a vector to a vector sum. If w is a vector sum, then we define $\mathrm{w}(w) = w$. To simplify the notation, we allow for vector sums to be added to vectors without explicit typecasting. Thus if v is a vector and w is a vector sum, then $v + w = \mathrm{w}(v) + w$ and $w + v = w + \mathrm{w}(v)$.

As an example, suppose we have three mask sequences and two vectors as follows

$$
\begin{aligned}
\mathbf{M}_0 &= \langle\langle 0, \langle 1 \rangle\rangle\rangle \ , \\
\mathbf{M}_1 &= \langle\langle 0, \langle 2, 2, 1, 2, 2, 2, 1 \rangle\rangle\rangle \ , \\
\mathbf{M}_2 &= \langle\langle 0, \langle 2, 2, 1, 2, 2, 2, 1 \rangle\rangle, \langle 0, \langle 2, 2, 3 \rangle\rangle\rangle \ , \\
v_1 &= \langle 1, 1, \mathbf{M}_0, \mathbf{M}_1 \rangle \ \text{ and} \\
v_2 &= \langle 1, 1, \mathbf{M}_0, \mathbf{M}_2 \rangle \ .
\end{aligned}
$$

If we now translate the note $\langle 0, 0 \rangle$ by v_1, then we get the note $\langle 1, 2 \rangle$, that is $\mathrm{T}(\langle 0, 0 \rangle, v_1) = \langle 1, 2 \rangle$. However, we cannot then translate $\langle 1, 2 \rangle$ by v_2 because $\langle 1, 2 \rangle$ is not in the space in which v_2 is defined. If n is a note and w is a vector sum, then the function $\mathrm{T}(n, w)$ in Figure 9 can be used to compute the note that results when n is translated by w. $\mathrm{T}(\langle 0, 0 \rangle, v_1 + v_2)$ is therefore undefined, whereas $\mathrm{T}(\langle 0, 0 \rangle, v_2 + v_1) = \langle 2, 5 \rangle$, illustrating the fact that vector addition is not commutative in this context.

3.5 Vector sequences, vector sum sequences and vector sum sets

A *vector sequence* is an ordered set of vectors and a *vector sum sequence* is an ordered set of vector sums. For any given vector sequence or vector sum sequence there exists an equivalent *vector sum set* which can be computed using the function in Figure 10. Any run of k identical vectors, v, in a vector

```
T(n, w)
1   n_2 ← n
2   for i ← 0 to |w| − 1
3       n_2 ← T(n_2, w[i])
4   return n_2
```

Figure 9: The function $\mathrm{T}(n, w)$ where n is a note and w is a vector sum.

```
S(V)
1   if V = ⟨⟩ return {}
2   w ← w(V[0])
3   W ← {w}
4   for i ← 1 to |V| − 1
5       w ← w + V[i]
6       W ← W ∪ {w}
7   return W
```

Figure 10: Function for computing the equivalent vector sum set, W, for $\mathcal{V}$, where $\mathcal{V}$ is either a vector sequence or a vector sum sequence.

sequence can be encoded as kv. For example, $\langle v_1, 3v_2, v_3\rangle = \langle v_1, v_2, v_2, v_2, v_3\rangle$. A run of k identical vector sums, w, in a vector sum sequence can similarly be encoded as kw.

If $\mathcal{V}$ is a vector set, vector sum set, vector sequence or vector sum sequence, then the function $\mathrm{W}(\mathcal{V})$, defined in Figure 11 returns the vector sum set that is equivalent to $\mathcal{V}$.

A single note, n, or a set of notes, N, can be used to generate a set of notes by translating it by a vector set, a vector sum set, a vector sequence or a vector sum sequence. Figures 12 and 13 show functions for carrying out these types of translation. A vector sum set therefore acts as a generalization of Deutsch and Feroe's concept of a "structure". Moreover, the combination of a note and a vector sum set to generate a set of notes is a generalization of Deutsch and Feroe's notion of combining a structure with a reference pitch to generate a sequence of pitches.

3.6 Product function

The *product function*, $\mathrm{P}(\mathbf{X})$, defined in Figure 14 is a generalization of Deutsch and Feroe's "prime" sequence operator, "pr". The single argument, $\mathbf{X}$, is an ordered set in which each element is a vector set, vector sum set, vector sequence or vector sum sequence. The first step in $\mathrm{P}(\mathbf{X})$ is to convert each element $\mathbf{X}[i]$ into its equivalent vector sum set (lines 1–3). The zero vector sum is then included in each vector sum set, $\mathbf{Y}[i]$ (lines 4–6). $\mathrm{P}(\mathbf{X})$ then returns a set containing a vector sum for each element of the n-ary Cartesian product of the vector sum sets in $\mathbf{Y}$ (lines 7–14). Note that if $\mathbf{A}$ and $\mathbf{B}$ are sequences such that $\mathbf{A} = \langle a_1, a_2, \ldots, a_m\rangle$ and $\mathbf{B} = \langle b_1, b_2, \ldots, b_n\rangle$ then $\mathbf{A} \oplus \mathbf{B} = \langle a_1, a_2, \ldots, a_m, b_1, b_2, \ldots, b_n\rangle$.

```
W(V)
1   if V is a vector sequence or vector sum sequence
2       W ← S(V)
3   else if V is a vector set
4       W ← ∅
5       for each v ∈ V
6           W ← W ∪ {w(v)}
7   else
8       W ← V
9   return W
```

Figure 11: Function for computing the equivalent vector sum set, W, for $\mathcal{V}$, where $\mathcal{V}$ is a vector sequence, a vector sum sequence, a vector set or a vector sum set.

```
T(n, V)
1   W ← W(V)
2   N ← ∅
3   for each w ∈ W
4       N ← N ∪ {T(n, w)}
5   return N
```

Figure 12: Function for translating a note, n, by a vector set, vector sequence, vector sum set or vector sum sequence, $\mathcal{V}$.

```
T(N, V)
1   N2 ← ∅
2   for each n ∈ N
3       N2 ← N2 ∪ T(n, V)
4   return N2
```

Figure 13: Function for translating a note set, N, by a vector set, vector sequence, vector sum set or vector sum sequence, $\mathcal{V}$.

```
P(X)
1   Y ← ⟨⟩
2   for i ← 0 to |X| − 1
3       Y ← Y ⊕ ⟨W(X[i])⟩
4   v0 ← ⟨0, 0⟩
5   for i ← 0 to |Y| − 1
6       Y[i] ← Y[i] ∪ {w(v0)}
7   W1 ← Y[0]
8   for i ← 1 to |Y| − 1
9       W2 ← ∅
10      for each w1 ∈ W1
11          for each w2 ∈ Y[i]
12              W2 ← W2 ∪ {w1 + w2}
13      W1 ← W2
14  return W1
```

Figure 14: The product function, $\mathrm{P}(\mathbf{X})$, where $\mathbf{X}$ is an ordered set in which each element is a vector set, vector sum set, vector sequence or vector sum sequence.

```
R_p(w)
1   w_2 ← w(R_p(w[0]))
2   for i ← 1 to |w| − 1
3       w_2 ← w_2 + R_p(w[i])
4   return w_2

R_t(w)
1   w_2 ← w(R_t(w[0]))
2   for i ← 1 to |w| − 1
3       w_2 ← w_2 + R_t(w[i])
4   return w_2
```

Figure 15: Functions for reflection. w is a vector sum.

```
R_p(V)
1   W_1 ← W(V)
2   W_2 ← ∅
3   for each w ∈ W_1
4       W_2 ← W_2 ∪ {R_p(w)}
5   return W_2

R_t(V)
1   W_1 ← W(V)
2   W_2 ← ∅
3   for each w ∈ W_1
4       W_2 ← W_2 ∪ {R_t(w)}
5   return W_2
```

Figure 16: Functions for reflection. $\mathcal{V}$ is a vector set, vector sum set, vector sequence or vector sum sequence.

3.7 Reflection

Deutsch and Feroe's "ret" and "inv" sequence operators correspond to reflection in the geometric language proposed here: "ret" corresponds to reflection in an axis parallel to the pitch axis, while "inv" corresponds to reflection in an axis parallel to the time axis. The functions,

$$\mathrm{R_p}(v) = \langle \mathrm{t}(v), -\mathrm{p}(v), \mathbf{M}_\mathrm{t}(v), \mathbf{M}_\mathrm{p}(v) \rangle \text{ and}$$
$$\mathrm{R_t}(v) = \langle -\mathrm{t}(v), \mathrm{p}(v), \mathbf{M}_\mathrm{t}(v), \mathbf{M}_\mathrm{p}(v) \rangle$$

reflect vectors in the time axis and pitch axis, respectively. The functions in Figure 15 can be used to reflect vector sums and the functions in Figure 16 can be used to reflect sequences or sets of vectors or vector sums. Note that there is no necessity for a function analogous to Deutsch and Feroe's "alt", because the music is represented as a *set* of geometrical *points* rather than a *sequence* of *symbols*. There is also no need for a scaling function to represent augmentation or diminution, since this can be accomplished using appropriate time mask sequences.

3.8 Examples

Figure 17 shows one of the examples used by Deutsch and Feroe [4, p. 504] to illustrate their model. This pattern can be encoded as $\mathrm{T}(n_1, \mathrm{P}(\langle 3v_1 \rangle, \langle v_2 \rangle))$ where

$$\begin{aligned} n_1 &= \langle 1, 60 \rangle \,, \\ v_1 &= \langle 1, 1, \mathbf{M}_1, \mathbf{M}_2 \rangle \,, \\ v_2 &= \langle -1, -1 \rangle \,, \\ \mathbf{M}_1 &= \langle \langle 1, \langle 2 \rangle \rangle, \langle 0, \langle 3 \rangle \rangle \rangle \text{ and} \\ \mathbf{M}_2 &= \langle \langle 0, \langle 2, 2, 1, 2, 2, 2, 1 \rangle \rangle, \langle 0, \langle 2, 2, 3 \rangle \rangle \rangle \,. \end{aligned}$$

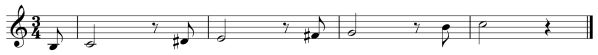


Figure 17: Example pattern used by Deutsch and Feroe [4, p. 504].

Given that the function, $\mathrm{U}(S_1, S_2, \ldots S_k)$ returns the union of sets $S_1, S_2, \ldots S_k$, then the pattern in Figure 4 can be encoded as follows:

$$\mathrm{U}(\mathrm{T}(n_1, \mathrm{P}(\langle v_1\rangle, \langle v_2\rangle)), \mathrm{T}(n_2, \mathrm{P}(\langle v_3\rangle)), \mathrm{T}(n_3, \mathrm{P}(\langle 2v_1\rangle, \langle v_2\rangle)))$$

where

$$\begin{aligned}
n_1 &= \langle 0, 90\rangle\ , \\
n_2 &= \langle 4, 87\rangle\ , \\
n_3 &= \langle 6, 85\rangle\ , \\
v_1 &= \langle 1, -1, \mathbf{M}_1, \mathbf{M}_2\rangle\ , \\
v_2 &= \langle 1, 1, \mathbf{M}_3, \mathbf{M}_2\rangle\ , \\
v_3 &= \langle 1, 1, \mathbf{M}_3, \mathbf{M}_4\rangle\ , \\
\mathbf{M}_1 &= \langle\langle 0, \langle 2\rangle\rangle\rangle\ , \\
\mathbf{M}_2 &= \langle m_1, \langle 0, s_1\rangle\rangle\ , \\
\mathbf{M}_3 &= \langle\langle 0, \langle 1\rangle\rangle\rangle\ , \\
\mathbf{M}_4 &= \langle m_1, \langle 3, s_1\rangle\rangle\ , \\
m_1 &= \langle 6, \langle 2, 2, 1, 2, 2, 2, 1\rangle\rangle \text{ and} \\
s_1 &= \langle 2, 2, 3\rangle\ .
\end{aligned}$$

Figure 18 shows bars 320–322 from the first movement of Beethoven's Sonata in E♭, Op. 7. This passage can be encoded as $\mathrm{U}(A, B)$ where

$$\begin{aligned}
A &= \mathrm{T}(n_1, \mathrm{P}(\langle 2v_1\rangle, \langle 5v_2\rangle))\ , \\
B &= \mathrm{T}(n_2, \mathrm{P}(\langle 17\langle 1, 0\rangle\rangle))\ , \\
n_1 &= \langle 0, 65\rangle\ , \\
n_2 &= \langle 0, 58\rangle\ , \\
v_1 &= \langle 0, 1, \mathbf{M}_1, \mathbf{M}_2\rangle\ , \\
v_2 &= \langle 1, -1, \mathbf{M}_1, \mathbf{M}_2\rangle\ , \\
\mathbf{M}_1 &= \langle 0, \langle 3\rangle\rangle \text{ and} \\
\mathbf{M}_2 &= \langle\langle 3, \langle 2, 2, 1, 2, 2, 2, 1\rangle\rangle, \langle 6, \langle 2, 2, 3\rangle\rangle\rangle\ .
\end{aligned}$$

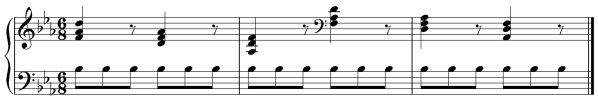


Figure 18: Bars 320–322 of Beethoven's Sonata in E♭, Op. 7, 1st. mvt.

4 MEL23: A PROTOTYPE IMPLEMENTATION OF MEL

MEL23 is the current version of a prototype implementation of the MEL language. The prototype takes the form of a MEL23 decoder (or compiler), that, when run, takes a MEL23 encoding as input and decodes this encoding to produce an in extenso description of the generated musical object in the form of a set of notes in note space. This set of notes can then be played or displayed visually. Figure 19 shows, on the left, a simple MEL23 encoding and, on the right, a visualization of the set of notes generated by this encoding.

As illustrated in Figure 19, a MEL23 encoding consists of a list of *statements* separated by semicolons. The first statement in a MEL23 encoding must be the *language identifier*, `MEL23;`. Each statement can

```
1  MEL23;
2  add(note(0,64),note(1,62),note(2,60));
3  play(50);
4  draw();
```

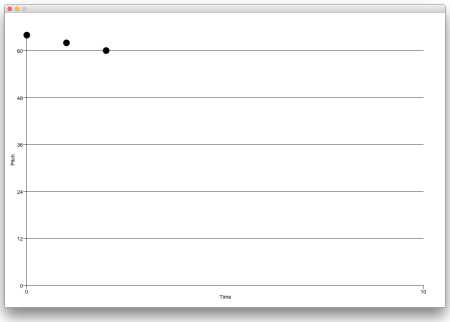


Figure 19: On the left, a simple MEL23 program that generates a note set containing three notes; on the right, a visualization of the note set generated by the program on the left.

either be a *function call* or an *assignment*. The statements in lines 2–4 of Figure 19 are all function calls. Each function call has the form

```
selector(arg1, arg2, ...)
```

where `selector` is the name (or selector) of the function and `arg1`, `arg2`, etc. are the arguments of the function. The arguments of a function may themselves be function calls. A function always returns a value, which may be `void`. If a function returns a non-void value, then it can be used in an assignment. An assignment statement always has the form

```
var = selector(arg1,arg2,...);
```

where `var` is the name of a variable. Variables do not need to be declared and can be referred to only after they have been assigned a value.

The example encoding in Figure 19 uses four functions: `add`, `note`, `play` and `draw`. The `add` function takes a list of notes or note sets and adds these to the *universal note set*, $\mathcal{U}$. In the example, three `Note` objects are added in line 2, each one constructed by calling the `note` function, which simply takes two arguments, an onset time followed by a MIDI note number, and returns a new `Note` object with those properties. The `play` function plays the musical object represented by the current value of $\mathcal{U}$. It takes one argument, which is the duration in milliseconds of one tatum. The `draw` function generates a visualization of the current value of $\mathcal{U}$ and displays it in a new window, as shown on the right in Figure 19.

Figure 20 lists all the functions defined in MEL23. As already explained, the `add` functions (lines 1–2) add their arguments to $\mathcal{U}$. The `coords` function (line 3) returns a `Coords` object, which can be used to specify the time and pitch component values of a note or a vector, without specifying any particular space. This is useful for defining MEL equivalents of the 'next', 'predecessor' and 'same' operators in the models of Deutsch and Feroe [4] and Simon and Sumner [18, 19]. Figure 21 shows a MEL23 encoding that assigns values to named variables (`nn`, `np`, etc.) that can be used in MEL in a way analogous to Deutsch and Feroe's 'next', 'predecessor' and 'same' operators. Assuming this encoding is stored in a file called "nps.mel", then these can be loaded at the start of any other encoding by using the `include` function (see Figure 20, line 5) which takes the name of the file to be included as its argument. Figure 22 shows how the assignments in Figure 21 can be used to encode Deutsch and Feroe's example pattern in Figure 17.

The encoding shown in the upper part of Figure 22 corresponds directly to that given in § 3.8 for this pattern. In line 2 of this encoding, the nps.mel file is loaded containing the definitions given in Figure 21. Then, in line 3, a time mask sequence, `M1`, is defined, which corresponds to the mask sequence, $\mathbf{M}_1 = \langle\langle 1, \langle 2\rangle\rangle, \langle 0, \langle 3\rangle\rangle\rangle$ in the encoding for this pattern in § 3.8 above. This definition uses the `mask` and `maskSequence` functions, which construct and return a new mask and mask sequence, respectively. Note that the `mask` function is polymorphic (see lines 7–8 in Figure 20). In the form used in Figure 22, the first argument specifies the offset and the second and subsequent arguments specify the mask structure. In lines 6 and 7 of the encoding shown in Figure 22, the `nn` and `pp` `Coords` objects are used as arguments to the `vector` function to specify two vectors, `v1` and `v2`. `v1` is a vector whose pitch mask sequence, `M2` (defined in line 4), represents a tonic triad in C major. The time mask sequence for this

```
add(Note...) : void
add(NoteSet...) : void
coords(Integer, Integer) : Coords
draw() : void
include(String) : void
listFunctions() : void
mask(Integer, Integer...) : Mask
mask(Integer, MaskStructure) : Mask
maskSequence(Mask...) : MaskSequence
maskStructure(Integer...) : MaskStructure
note(Integer, Integer) : Note
noteSet(Note...) : NoteSet
play(Integer) : void
print() : void
product(Object...) : VectorSumSet
reflectPitch(Vector) : Vector
reflectPitch(VectorCollection) : VectorSumSet
reflectPitch(VectorSum) : VectorSum
reflectTime(Vector) : Vector
reflectTime(VectorCollection) : VectorSumSet
reflectTime(VectorSum) : VectorSum
repeat(Integer, Vector) : VectorSequence
repeat(Integer, VectorOrVectorSum) : VectorSumSequence
setDifference(NoteSet, Note...) : NoteSet
space(MaskSequence, MaskSequence) : Space
translate(Note, Vector) : Note
translate(Note, VectorCollection) : NoteSet
translate(Note, VectorSum) : Note
translate(NoteSet, Vector) : NoteSet
translate(NoteSet, VectorCollection) : NoteSet
translate(NoteSet, VectorSum) : NoteSet
union(NoteSet...) : NoteSet
union(VectorCollection...) : VectorSumSet
vector(Coords) : Vector
vector(Coords, MaskOrMaskSequence, MaskOrMaskSequence) : Vector
vector(Coords, Space) : Vector
vector(Integer, Integer) : Vector
vector(Integer, Integer, MaskOrMaskSequence, MaskOrMaskSequence) : Vector
vector(Integer, Integer, Space) : Vector
vectorAdd(VectorCollection, VectorCollection) : VectorSumSet
vectorAdd(VectorCollection, VectorOrVectorSum) : VectorSumSet
vectorAdd(VectorOrVectorSum, VectorCollection) : VectorSumSet
vectorSequence(Vector...) : VectorSequence
vectorSet(Vector...) : VectorSet
vectorSum(VectorOrVectorSum...) : VectorSum
vectorSumSet(VectorCollection) : VectorSumSet
vectorSumSet(VectorOrVectorSum) : VectorSumSet
vectorSumSet(VectorOrVectorSum...) : VectorSumSet
```

Figure 20: Functions defined in MEL23.

```
MEL23;
nn = coords(1,1);
np = coords(1,-1);
ns = coords(1,0);
pn = coords(-1,1);
pp = coords(-1,-1);
ps = coords(-1,0);
sn = coords(0,1);
sp = coords(0,-1);
ss = vector(0,0);
```

Figure 21: A MEL23 encoding that defines multidimensional equivalents of Deutsch and Feroe's 'next', 'same' and 'predecessor' operators.

```
1  MEL23;
2  include("nps");
3  M1 = maskSequence(mask(1,2),mask(0,3));
4  M2 = maskSequence(mask(0,2,2,1,2,2,2,1),mask(0,2,2,3));
5  n1 = note(1,60);
6  v1 = vector(nn,M1,M2);
7  v2 = vector(pp);
8  add(translate(n1,product(repeat(3,v1),v2)));
9  play(100);
10 draw();
```

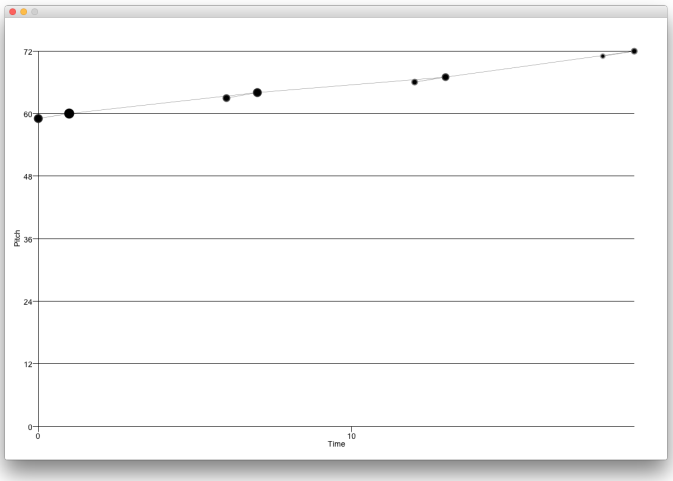


Figure 22: Using the definitions in Figure 21 to encode Deutsch and Feroe's example pattern (Figure 17) in MEL23.

vector represents a rhythm consisting only of time points falling at the beginnings of bars in 3/4 (with the start of the first bar occurring at time point 1). A step to the next element in the space defined by these two mask sequences therefore corresponds to moving to the next pitch in a C major triad and to the time point at the beginning of the next 3/4 bar. This is accomplished using the `nn Coords` object which is defined to be equal to `coords(1,1)` (see Figure 21, line 2), which is given in line 6 of Figure 22 as the first argument to the `vector` constructor function. Similarly, the `pp Coords` object is used in line 7 to define a vector that goes to the previous pitch and the previous time point in note space (if no space is given to the `vector` function, then the vector returned is in note space by default). Finally, in line 8, the function call

```
translate(n1,product(repeat(3,v1),v2))
```

implements the expression

$$\mathrm{T}(n_1, \mathrm{P}(\langle 3v_1 \rangle, \langle v_2 \rangle))$$

given in § 3.8 above as a possible encoding for this pattern. As this example clearly shows, the various forms of the `translate` function (lines 26–31 in Figure 20) implement the function T, defined in Figures 12 and 13. The product function defined in Figure 14 is implemented in the `product` function in MEL23 (line 15 of Figure 20) and the notation for encoding runs of identical vectors or vector sequences is implemented using the two forms of the `repeat` function (lines 22–23 in Figure 20).

The second example given in § 3.8 above is an encoding of the analysis in Figure 4. The implementation of this encoding in MEL23 is shown in Figure 23. Note that this encoding makes use of the fact that mask structures can be defined independently of any specific masks (see definition and use of the mask structure, `s1`, in lines 3, 6 and 8 in Figure 23). The encoding also uses the `np Coords` object (line 12), which represents a step to the **n**ext time point and the **p**revious element of the current pitch alphabet. Note also that, in this figure, as in Figures 19 and 22, the lines drawn between the points in the visualization of the generated $\mathcal{U}$, provide a trace of how the notes are generated by the MEL encoding. Notes represented by larger dots are generated earlier in the process by which the encoding produces the musical object.

An implementation of the third example encoding in § 3.8 (Figure 18) is shown in Figure 24. In this example, the `sp Coords` object is used (line 7), which represents a repeat of the same time point and a

```
MEL23;
include("nps");
s1 = maskStructure(2,2,3);
m1 = mask(6,2,2,1,2,2,2,1);
M1 = maskSequence(mask(0,2));
M2 = maskSequence(m1,mask(0,s1));
M3 = maskSequence(mask(0,1));
M4 = maskSequence(m1,mask(3,s1));
n1 = note(0,90);
n2 = note(4,87);
n3 = note(6,85);
v1 = vector(np,M1,M2);
v2 = vector(nn,M3,M2);
v3 = vector(nn,M3,M4);
add(translate(n1,product(v1,v2)),
    translate(n2,product(v3)),
    translate(n3,product(repeat(2,v1),v2)));
play(100);
draw();
```

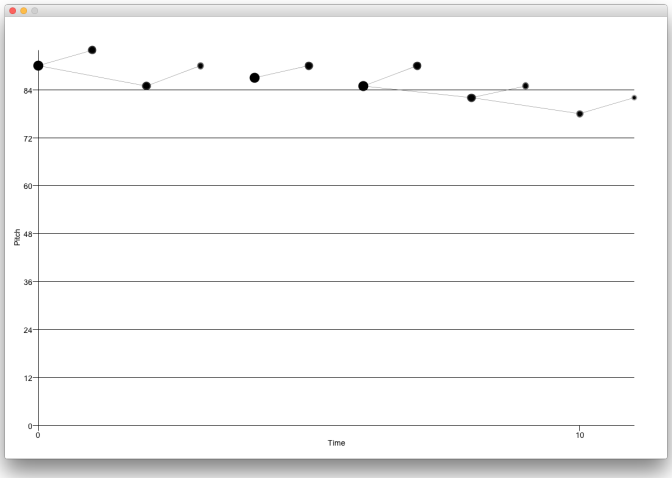


Figure 23: An implementation in MEL23 of the encoding given in § 3.8 for the analysis shown in Figure 4.

```
MEL23;
include("nps");
n1 = note(0,65);
n2 = note(0,58);
M1 = maskSequence(mask(0,3));
M2 = maskSequence(mask(3,2,2,1,2,2,2,1),mask(6,2,2,3));
v1 = vector(sp,M1,M2);
v2 = vector(np,M1,M2);
A = translate(n1,product(repeat(2,v1),repeat(5,v2)));
B = translate(n2,product(repeat(17,vector(ns))));
add(A,B);
play(100);
```

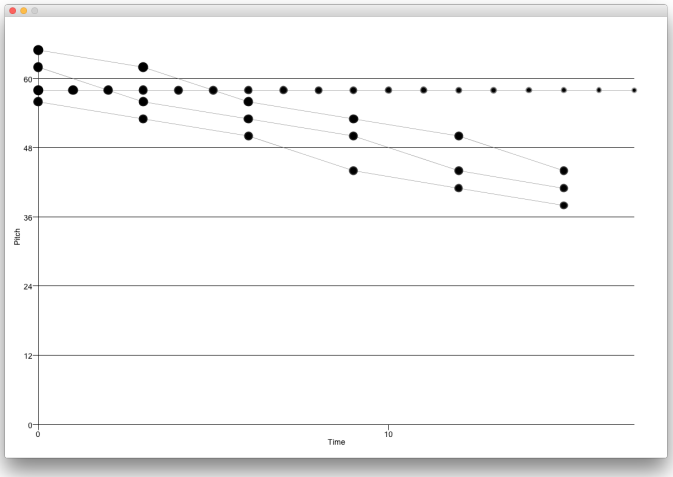


Figure 24: An implementation in MEL23 of the encoding given in § 3.8 for the Beethoven extract shown in Figure 18.

move to the previous element in the current pitch alphabet.

5 SUMMARY

This report has presented the fundamental concepts of MEL, a geometric music encoding language, designed to permit the structures of polyphonic musical objects to be described parsimoniously. Perhaps the most important feature of the language is the way in which it extends Deutsch and Feroe's concept of an alphabet and generalizes it to the time dimension, allowing for repeated rhythms and metrical structures as well as pitch structures such as scales and chords to be represented in an identical way.

The current prototype implementation of MEL, MEL23, has also been presented and some examples have been given of how specific ways of understanding musical objects can be encoded in this language. The MEL23 implementation consists of a Java class that, when run, takes a MEL23 encoding as input and decodes or compiles this encoding to produce a set of notes as output, which can then be either printed out, played or displayed visually.

References

[1] I. Bent. *Analysis*. The New Grove Handbooks in Music. Macmillan, 1987. (Glossary by W. Drabkin).

[2] G. J. Chaitin. On the length of programs for computing finite binary sequences. *Journal of the Association for Computing Machinery*, 13(4):547–569, 1966.

[3] N. Chater. Reconciling simplicity and likelihood principles in perceptual organization. *Psychological Review*, 103(3):566–581, 1996.

[4] D. Deutsch and J. Feroe. The internal representation of pitch sequences in tonal music. *Psychological Review*, 88(6):503–522, 1981.

[5] K. Koffka. *Principles of Gestalt Psychology*. Harcourt Brace, 1935.

[6] A. N. Kolmogorov. Three approaches to the quantitative definition of information. *Problems of Information Transmission*, 1(1):1–7, 1965.

[7] E. L. J. Leeuwenberg. A perceptual coding language for visual and auditory patterns. *American Journal of Psychology*, 84(3):307–349, 1971.

[8] F. Lerdahl and R. S. Jackendoff. *A Generative Theory of Tonal Music*. MIT Press, 1983.

[9] M. Li and P. Vitányi. *An Introduction to Kolmogorov Complexity and Its Applications*. Springer, third edition, 2008.

[10] D. Meredith. A geometric language for representing structure in polyphonic music. In *Proceedings of the 13th International Society for Music Information Retrieval Conference (ISMIR 2012)*, Porto, Portugal, 2012.

[11] D. Meredith. Music analysis and Kolmogorov complexity. In *Proceedings of the 19th Colloquio di Informatica Musicale (XIX CIM)*, Trieste, Italy, 2012.

[12] D. Meredith. Analysing music with point-set compression algorithms. In D. Meredith, editor, *Computational Music Analysis*, pages 335–366. Springer, 2015.

[13] D. Meredith, K. Lemström, and G. A. Wiggins. Algorithms for discovering repeated patterns in multidimensional representations of polyphonic music. *Journal of New Music Research*, 31(4):321–345, 2002.

[14] M. T. Pearce. *The Construction and Evaluation of Statistical Models of Melodic Structure in Music Perception and Composition*. PhD thesis, Department of Computing, City University London, 2005.

[15] F. Restle. Theory of serial pattern learning: Structural trees. *Psychological Review*, 77(6):481–495, 1970.

[16] J. Rissanen. Modeling by shortest data description. *Automatica*, 14(5):465–471, 1978.

[17] H. A. Simon. Complexity and the representation of patterned sequences of symbols. *Psychological Review*, 79(5):369–382, 1972.

[18] H. A. Simon and R. K. Sumner. Pattern in music. In B. Kleinmuntz, editor, *Formal representation of human judgment*. Wiley, New York, 1968.

[19] H. A. Simon and R. K. Sumner. Pattern in music. In S. M. Schwanauer and D. A. Levitt, editors, *Machine Models of Music*, pages 83–110. MIT Press, Cambridge, MA, 1993. Published earlier as [18].

[20] R. J. Solomonoff. A formal theory of inductive inference (Part I). *Information and Control*, 7(1):1–22, 1964.

[21] R. J. Solomonoff. A formal theory of inductive inference (Part II). *Information and Control*, 7(2):224–254, 1964.

[22] D. Temperley. *Music and Probability*. MIT Press, 2007.

[23] P. A. van der Helm and E. L. Leeuwenberg. Accessibility: A criterion for regularity and hierarchy in visual pattern codes. *Journal of Mathematical Psychology*, 35:151–213, 1991.

[24] H. L. F. von Helmholtz. *Treatise on Physiological Optics*. Dover, New York, 1910/1962. Trans. and ed. by J. P. Southall. Originally published in 1910.