Towards a calculus for expressive timing in music

Peter Desain & Henkjan Honing

[published as: Desain, P., & Honing, H. (1991). Towards a calculus for expressive timing in music. Computers in Music Research, 3, 43-120.]

© copyright 1992, Peter Desain & Henkjan Honing

CONTENTS

Contents 2
Introduction 1
Overview of the calculus 3
Characteristics 3
Representation 5
Implementation 6
Musical objects 8
Basic musical objects 8
Structured musical objects 9
Multilateral structures 9
Collateral structures (ornamented objects) 10
S, a multilateral successive structure 11
P, a multilateral simultaneous structure 12
APPOG, a collateral successive structure 13
ACCIA, a collateral simultaneous structure 14
Example of the representation of a musical object 15
Representing expression 16
Expressive tempo 16
Expressive asynchrony 16
Expressive articulation 17
Definition of articulation 17
Estimate onsets 18
Articulation invariance 18
Expression maps 18
Onset timing 18
Articulation expression 20
Operations on expression maps 21
Scale maps 21
Scaling expressive tempo 21
Scaling the expressive tempo of an S section 22
Scaling the expressive tempo of an APPOG section 24
Scaling expressive asynchrony 25
Scaling the expressive asynchrony of a P section 26
Scaling the expressive asynchrony of an ACCIA section 28
Scaling expressive articulation 29
Scaling the expressive articulation of a multilateral section 30
Scaling the expressive articulation of a collateral section 32
Keeping articulation consistent in the scaling of expressive timing 33
Stretch maps 35
Interpolate maps 35
Transfer maps 35
Transformations 35
Scale timing 37
Keeping articulation consistent 41
Scale intervoice expression 41
Conclusion 43
Acknowledgements 43
References 43
Microworld expression calculus 45

Towards a calculus for expressive timing in music

Peter Desain & Henkjan Honing

NICI
Nijmegen University
P.O.Box 9104
NL-6500 NE Nijmegen
Netherlands

University of Amsterdam
Computational Linguistics
Spuistraat 134
NL - 1012 VB Amsterdam
Netherlands

This paper presents a calculus that enables expressive timing to be transformed on the basis of the structural aspects of the music. Expression within a unit is defined as the deviations of its parts with respect to the norm set by the unit itself. The behaviour of musical material under expressive transformations is determined uniquely by its structural description and the type of expression. Although the calculus separates different kinds of behaviour, it entails no musical knowledge of the transformations themselves and it also does not model music cognition. The algorithmic simplicity of the calculus combined with its elaborate knowledge representation mirrors the common hypothesis that the complex expressive timing profiles found in musical performances can be explained as the product of a small collection of simple rules linked to a relatively complex structure. The calculus will hopefully prove to be a solid basis for formalised theories of music cognition.

INTRODUCTION

In Desain and Honing (1992a) we argued that a simplistic notion of a tempo curve of a musical performance is a dangerous and harmful theoretical construct. Although the use of a tempo curve to describe time measurements is perfectly sound, the notion itself is often presented as a cognitive or musical concept. And tempo curves do not have any right to exist in those domains. In the above article, this was concluded from the fact that when it is used as a basis of transformations, inevitably the results make no musical sense. The cause of this failure can often be attributed to the lack of structural information in the tempo curve. For example, in changing the overall tempo of a performance, by manipulating the tempo curve alone, all time intervals of equal length between two notes are scaled in the same way. But some notes may constitute a particular kind of ornamentation, whose duration should be more or less unaffected by tempo. As a result the timing of the piece becomes unmusical. And there are many more examples of transformations that cannot be done on isolated tempo curves. Because the article had an essentially negative tone - identifying the problems and their causes - we felt compelled to follow it up with a study of possible solutions.

This paper is an attempt to identify ways in which structural knowledge can be used to enable expression transformations on musical performances that do make musical sense.
In past research we considered expression merely as deviations of attributes of performed notes from their value notated in a score. This definition, however useful in the initial study of expressive timing, soon lost its attractiveness. In general, listeners can appreciate expression in music performance without knowing the score. And a full reconstruction of the score in the form of a mental representation is often impossible. Take for instance the notion of loudness of notes. Should a listener be required to fully reconstruct the dynamic markings in the score before it is possible to appreciate the deviations from this norm as expressive information added by the performer? Such a nonsensical conjecture indeed follows from a rigid definition of expression as deviations from the score. But it is possible to find ways of defining expression on the basis of performance information only. The more so since it became possible to model the quantization of performed note durations into discrete categories (Desain & Honing, 1991), and therefore even the extraction of performed tempo is possible directly from the performance itself.

In this paper we will base expression on the notion of structural units in a working definition: expression within a unit is defined as the deviations of its parts with respect to the norm set by the unit itself. An example might make this more clear. Lets take, for instance, a metrical hierarchy of bars and beats; the expressive tempo within a bar can be defined as the pattern of deviations from the global bar tempo generated by the tempo of each beat. Or, take the loudness of the individual notes of a chord; the dynamic expression within a chord can be defined as the set of deviations from the mean chord loudness by the individual notes. Using this intrinsic definition, expression can be extracted from the performance data itself, taking more global measurements as reference for local ones, assuming that the structural units themselves are known. Thus the structural description of the piece becomes central, both to establish the units which will act as a reference, and to determine its subunits that will act as atomic parts whose internal detail will be ignored. A generalization of this concept can also deal with expression arising from the interplay of two or more voices.
It will be clear by now that any other connotations of the concept of musical expression, its link to human affect and extra-musical indexicality, however interesting, will be ignored here completely.

Before the details of the calculus are presented it might be fitting to give some explanation for undertaking for this work. First of all, we think that the research of expression in music is in need of measurement instruments that can cope with the enormous complexity of performance data and that are much more sophisticated than tempo

curves. Some of the proposed transformations can be used as an “auditory microscope” by exaggerating expression at certain structural levels, like amplifying the timing lead, the melody often has over the accompaniment. Some of the tools presented can be used as “expression scalpels” for trimming away certain kinds of expression that might obscure other phenomena, like removing the tempo deviations within each beat, but holding the timing patterns of the beats themselves invariant. Other tools can “transplant” musical expression from one piece of music to the other, say from a theme to its variation. The availability of this ‘machinery’ will deepen our understanding of the intricacies of music performance expression.

A further motivation is the practical applicability of this work in systems for computer music. Especially the music editors and sequencer programs that are commercially available nowadays which are in need of better ways to treat musical information in musical ways. Expressive timing should not be considered a nasty feature of performed music, as it is in nowadays multi-track recording techniques where tempo, timing and synchronization are treated as technical problems. Instead expressive timing has to be regarded as an integral quality of performed music whereby the performer communicates structural aspects of the music to the listener (Clarke, 1988). We hope that our work can inspire new music software based on this view.

OVERVIEW OF THE CALCULUS

Characteristics

The calculus has the following important characteristics:

The calculus is described here only <u>for different brands of expressive timing</u>. Dynamics could be formalised along the same lines, but for clarity we restrict ourselves to the domain of expressive timing. Other attributes that carry expression, like intonation, vibrato and timbre may require a different treatment.

The <u>types of expression have to be computable</u> to be within reach of this calculus. One must be able to calculate the expression at every level of the structural hierarchy, given the expression of their components (e.g. the timing of a chord must be computable when the timing of the embedded notes is given). One also must be able to state ways to effectively set the expression of the components once the expression of the whole is given (i.e. propagate a shift in timing down the hierarchy, to the basic objects carrying the expression). Types of expression that do not have this characteristic - or are not yet formalised as such- cannot be described.

Both <u>performance and “score” timing of individual notes are clearly defined</u>. Notes require attributes that can be measured more or less directly from the performance data like the note onset time and the offset time. At least the onset time must be clearly

specified, which makes the calculus less appropriate for expressive performances by instruments for which onset times are not so clear cut. Secondly, the metrical note duration (the timing of the note as notated in the score) must also be available as a note attribute - either via quantization or by matching a performance to a known score. These processes are considered preprocessing here. Although the reference to score duration, score onset and score offset times is less appropriate in the context of our definition of expression - we will use this terminology, for lack of better terms.

The "score" timing of rests is clearly defined. Perhaps surprisingly, the rest plays a key role in some transformations. So we assume that it either can be inferred from the performance timing (Longuet-Higgins, 1976 shows a way of doing so), or it is recovered via the matching of a performance and a known score.

All proposed transformations are structure preserving. This means that the calculus is restricted to true expressive transformations: the score timing of the notes is known and fixed, and transformations will leave this and the structural description invariant.

The behaviour of musical material under expressive transformations is determined uniquely by its structural description and the type of expression.

The transformations are defined on a hierarchical structural description uniquely linking all material. Ambiguous structural descriptions (e.g. two or more possible structural descriptions) or incomplete descriptions cannot be dealt with. The obvious need for knowledge representations containing multiple structural descriptions (metrical, phrase, and rhythmical grouping structures, different analysis etc.) is not denied. We just require that such representations be preprocessed to select only one complete structural description. This is not a real restriction since transformations based on different kinds of structural knowledge of the same piece can always be done in sequence. Re-inserting the trimmed structural descriptions into a transformed piece is trivial because the transformations preserve the structure.

Naturally, the higher-level structural description of the piece must be consistent with the performance timing. For example, a structural description of the piece in which two notes are given a certain sequential time order (one after the other) - can only fit a performance in which at least the onset of the corresponding notes obey the same order. The precise rules will be given when the structural descriptions are introduced.

The transformations are defined to apply to a certain level of the hierarchical structural description, ignoring details from lower levels and keeping higher levels invariant. Means to select such a level are assumed. In sophisticated realisations of the calculus

this may entail a match language ("the first bar of the piano solo that begins with a C") or a graphical representation. In this paper we will simply assume that each musical object has a name as attribute and defines a structural level as the set of objects with a certain name.

Although the calculus separates different kinds of behaviour, it entails no musical knowledge of the transformations themselves. Accordingly, the proposed knowledge representation does support for example, arbitrary descriptions of the metrical structure of a piece, but has no knowledge of "the best structural analysis". To give a second example: the proposed knowledge representation does support ways to modify timing (a)synchrony between voices, but it has no knowledge about correct or effective ways of using this in musical performance.

The calculus also does not model cognition. It does not state how, for example, voice-leading helps auditory streaming, how phrase final lengthening beyond a limited range disables rhythm perception, or how structure is communicated by the expressive timing profiles. However, this work constitutes a solid basis for formalised theories about these issues, providing a powerful representation in which they can be expressed.

Representation

Several concepts are used in the calculus:

Musical objects are either of a basic nature or form a structural description of a collection of musical objects. Basic musical objects consist of notes and rests. Notes are the only musical objects that carry the expressive information. Structural descriptions form collections of musical objects. They may describe hierarchical time intervals like metrical-, phrase- or rhythmical grouping, they can group the notes of chords and ornaments together, or form large horizontal slices through the piece, describing the separate voices etc. Mere collections (sets) of objects are too meager a basis for most transformations, therefore, structural descriptions specify the intended relations in time between these objects as well (Honing, 1992). Most transformations can be defined if two orthogonal characteristics of the structural description are given: the temporal nature and the ornamenting quality. The first describes whether a sequence or a parallelism (a so called successive or simultaneous construct) is represented. The second describes whether the musical object is considered an ornament attached to another object or not. Ornaments are shielded from certain modifications and refer to another object for certain attributes. These two binary characteristics result in four concrete types of structural description that will be described in detail later.

Expressive magnitudes are values of expressive measurement on a certain scale. The scales themselves are of course crucial in modeling effective transformations, in cognitive and musical senses. For example, a tempo scale on which a transformation to make something twice as fast actually yields a double perceived tempo is quite useful. But for the sake of simplicity we abstract from the perceptual processes and the instruments that generate the sound, and will just assume simple physical measurements of time and other expressive attributes.

Expression maps describe the expressive patterns of structured musical objects at a certain structural level. They consist of a section for each musical object at that level. A section lists the expressive values for all components of that object. They come in different brands - consistent with the type of musical structure where they were extracted from. Expression maps can be extracted from and applied to musical objects, with possibly a modification of the map in between.

Expression types are sets of procedures to extract a particular type of expression map from a musical object, to impose it on a musical object, and to modify the map. They capture the difference between expressive tempo, asynchrony, and articulation. They may become fairly sophisticated, like a brand of expressive tempo that knows how to keep the articulation of an individual note invariant when the timing of the note onsets is changed.

Modifications are defined as operations on expression maps. They may scale, interpolate, or do any other operation on the map. They are often designed such that certain characteristics are kept invariant, e.g. the total duration of a section while changing the timing of the parts.

Transformations are defined as operations on musical objects. They are often direct generalizations of the expression map modifications - first extracting the map, applying the modification and imposing the modified map. They also handle the selection of the level of structural description on which to apply the transformation. Furthermore, they may have means to maintain consistency among the affected level and other musical material, e.g. making an accompaniment obediently follow the transformation in expressive tempo applied to the melody.

Implementation

Part of the work described in this paper was done in the design of the POCO system (Honing, 1990) for which a scaling operation of expressive timing linked to structural descriptions was implemented. But, in evaluating this rather complex piece of software, better abstractions arose. Especially the design of a set of data structures for music that

capture the differences in behaviour under transformation proved beneficial. Which again illustrated the adage:

> *"Get your data structures correct first, and the rest of the program will write itself."* (David Jones, quoted in Bentley, 1988)

Because the constructs interact heavily, and because it should be easy to add unforeseen new constructs (like a new type of expression), musical objects, expression maps and expression types are implemented as classes in an object-oriented language. In that way it is easy to express modifications and transformations as polymorphic operations that will behave according to the type (the class) of their arguments. The slicing-up of knowledge in classes means answering questions like: which part of the extraction procedure of an expressive tempo map of a sequential musical object is specific for expressive tempo only and should be stated within the expression type; which part only depends on the sequential nature of the musical structure, and should be part of the class for sequential musical objects; and which part describes the creation of an expression map and belongs to that class?

Although a good Object-Oriented Language (we used CLOS, a Lisp-based system) provides one with the programming-constructs needed to express these concepts, the actual process of factoring knowledge into these polymorphic procedures is still a difficult one, especially because during the design of the best structure of the classes - allowing for the most elegant factoring of the procedures - cannot be completely foreseen. This forced us to go through several re-design rounds before the concepts stabilized in their present form.

The following CLOS (Keene, 1989; Steele, 1990) constructs were used heavily in the implementation: multiple inheritance (forming class dependencies that are more complex than simple hierarchies), multi-methods (functions that are polymorphic in more arguments), mix-in type of inheritance (grouping of partial behaviour in an abstract class that must be mixed in with other classes to supply that behaviour to their instances), method combination (providing ways of combining partial descriptions of behaviour of one method for more classes). Together they make it possible to extend the system by adding program code only, instead of rewriting it.

The calculus will be incorporated in POCO. The other tools available in POCO, like score-performance matchers, multiple structural descriptions, storage and retrieval from standard MIDI-files, playback and editors for music text formats etc., will support a comfortable use of the calculus with real performance data. An implementation in the form of the microworld is given in the appendix and aimes at conciseness and elegance. Luckily, this goal only occasionally conflicts with computational efficiency.

The following five paragraphs will describe the calculus in more detail. The reader interested in the more general aspects of the calculus is advised to continue reading below Transformations.

MUSICAL OBJECTS

Musical objects come in different brands. Some types are specific enough to describe an object completely (the instantiatable or concrete classes). Other types are used as a descriptive grouping of likewise behaviour (the abstract classes). The types of musical objects and their interrelations are shown in figure 1.

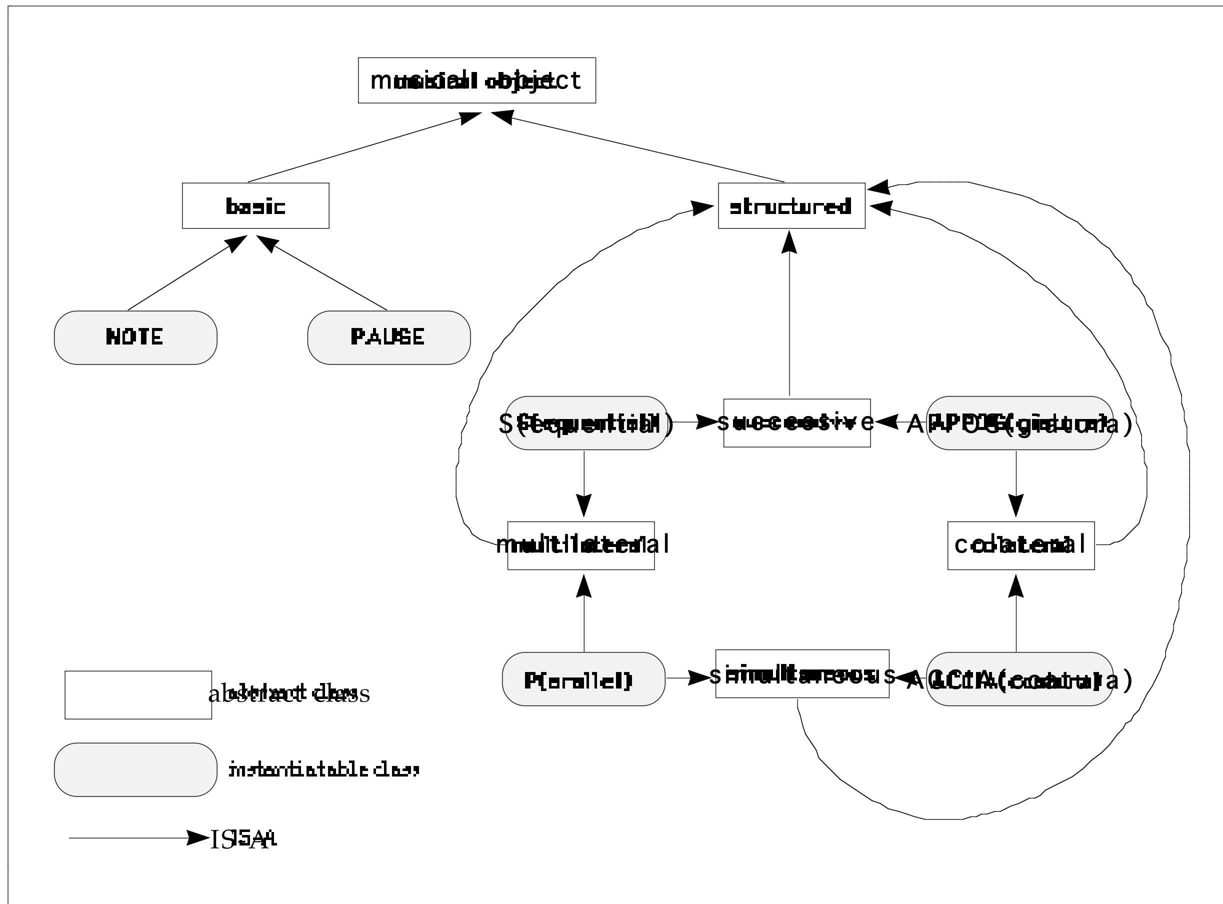

Figure 1. Classes of musical objects and their interrelations.

Basic musical objects

Basic musical objects are notes and rests (we use the word PAUSE to avoid the name clash with the Common Lisp primitive REST). In examples we will use notes with clearly observable onset and offset times (called PERF-ONSET and PERF-OFFSET) measured in ms. from the beginning of the performance. Both notes and rests have as a property a time position in the score (called SCORE-ONSET and SCORE-OFFSET) measured in any kind of (beat)-count (a rational number). These score times are calculated automatically from the supplied score durations of notes and rests via the structural descriptions. This facilitates easy creation of large scores.

Rests are essential and cannot just be ignored, as is done in some low-level representations (e.g. the Midi-file standard). They are central e.g. in dealing with articulation - a short

note followed by a rest behaves differently under transformation than a longer note played in a staccato way.

Structured musical objects

Multilateral structures

In research on music perception and cognition a distinction is often made between successive temporal processes that deal with events occurring one after another, and simultaneous temporal processes that handle events occurring around the same time (e.g. Bregman, 1990; Serafine, 1988). For the first type of events of the expressive means can be rubato - the change of tempo over the sequence. In the second one the expressive means can be chord-spread and asynchrony between voices, both more timbral aspects. These processes work differently in perception. Since we want to imply differences in behaviour mainly by differences in structural description a way should be found in which both these constructs can be represented.

We propose to use for this purpose the simple time structures S and P that functioned well in (Desain & Honing, 1988; Desain, 1990; Desain & Honing, 1992b). If a collection of musical objects is formed such that they occur one after another they are described as a successive structured object named S (for Sequential). If a collection of musical objects occur at the same time they can be collected in a simultaneous structured object called P (for Parallel). These structures serve as a general way to represent a collection together with the temporal relation between the components, as stated in the score. We call the objects *multilateral* because their components are considered to be of equal importance, and are to be treated as such in expressive transformations.

The score times of a structured object and its parts are constrained by consistency rules. They are described separately in frames 1 and 2. These constraints are enforced by specifying only notes and rests with a score duration. The constraints propagate these automatically when a structural description is created and set all score onset and offset times.

In calculating expression, the previous and subsequent context of musical objects is sometimes needed. For instance, consider articulation: possibly defined as the overlap between the sounding parts of a note and the next one, i.e. the time difference between the offset of the note itself and the onset of the “next” note. Besides “next material” a link to “previous material” is foreseen to be needed as well, e.g. in the calculation of local accent patterns. To formalize and generalize this notion of “previous” and “next” material a definition of the left and right context of a musical object is given. This notion also reflects the fact that some expressive values cannot be calculated because some contexts are not available or carry no expression e.g. the tempo of the last note in a piece, or the performance onset of a voice that starts with a rest. Expressive transformations must thus expect to come across missing values in an expression map. The notion of context is

explicitly represented in the program as attributes of the objects themselves. This is possible because the structural description is invariant and so are the contexts. Another possibility would be to represent them implicitly, recovering them by search via a bi-directional part-of link between musical objects. Alternatively, they could be represented tacitly, i.e. supplying them by a general control structure that walks through structured musical objects.

Collateral structures (ornamented objects)

Some musical objects contain components that should maintain a dependency relation to one another. If such a *collateral* pair is transformed, the transformation should be carried out on the main component only, the submissive one obediently following the main component's transformation, but not being transformed itself. An ornamented musical object like a graced note (a note preceded by a grace note), is a good example of a collateral object. For example, in the scaling of the expressive tempo of a melody which contains a graced note, the data on which the expressive transformation is carried out (in this case the performance onset) stems from the main object. The grace note is ignored. When in the actual transformation the graced note pair is stretched or compressed and moved to an other point in time, only the main note will undergo that operation. The ornament will just follow its shift in time.

A second use of this concept is made when the relation of an ornament to its main object, within such a collateral couple, is considered to be expressive, and a potential source of expressive transformations. In this case, the main object stays invariant, and only the ornament undergoes transformation. Take for example the asynchrony between the performance onset of a grace note and the note it is attached to. This time interval can be modified by appropriate means, resulting in local changes of the timing of the grace note - but keeping the timing of the main note invariant.

Collateral (ornamented) objects can again have two kinds of temporal nature: successive or simultaneous. The first one is called APPOG (for appoggiatura). It describes a "time-taking" ornament where the ornament is considered to finish when its main object starts (all in terms of score times). The second is called ACCIA (for acciaccatura). It can represent a so called "time-less" ornament that is supposed to start at the same time as the object it is attached to. Note that both parts of a collateral pair are musical objects themselves and can have internal structure. The concepts of APPOG and ACCIA ornamented objects are an elaboration of the PRE and POST objects that were introduced in (Desain & Honing, 1988). Consistency rules for score times and context are described in frames 3 and 4.

S, a multilateral successive structure

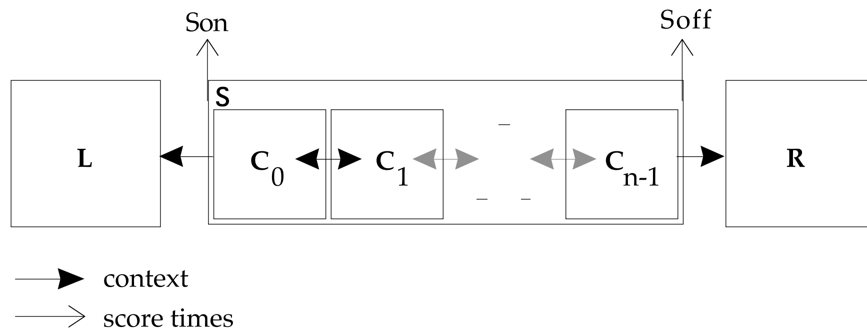

*Figure of **S** object*

Consider an S structure of n components C_i with $0 \le i \le n-1$.
Assume that component C_i has score onset time Son_i, score offset time $Soff_i$ and that the whole structure has score onset time Son and score offset time Soff. Then the following must hold:

$Son = Son_0$
$Soff = Soff_{n-1}$
$Soff_i = Son_{i+1}$, for $0 \le i \le n-2$

Assume that component C_i has performance onset time Pon_i and that the whole structure has performance onset time Pon. Then the following must hold:

$Pon = Pon_0$
$Pon_i \le Pon_{i+1}$, for $0 \le i \le n-2$

Assume that component C_i has left context L_i and right context R_i and that the whole structure has left context L and right context R. Then the following holds:

$L = L_0$
$R = R_{n-1}$
$R_i = C_{i+1}$, for $0 \le i \le n-2$
$L_i = C_{i-1}$, for $1 \le i \le n-1$

Frame 1. Description of a S structure.

P, a multilateral simultaneous structure

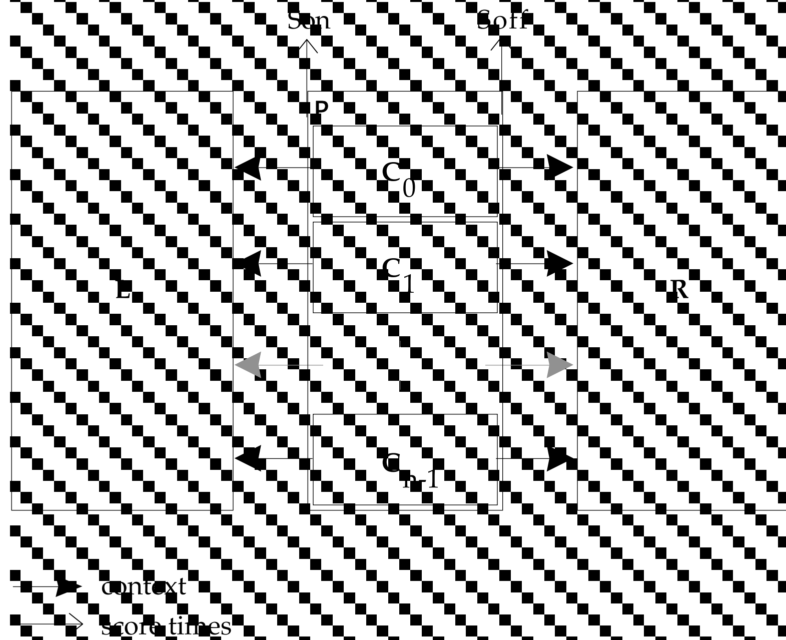

*Figure of **P** object*

Consider a P structure of n components C_i with $0 \leq i \leq n-1$.

Assume that component C_i has score onset time Son_i and score offset time $Soff_i$ and that the whole structure has score onset time Son and score offset time Soff. Then the following must hold:

$Son_i = Son$, for $0 \leq i \leq n-1$

$Soff_i = Soff$, for $0 \leq i \leq n-1$

Assume that component C_i has performance onset time Pon_i and that the whole structure has performance onset time Pon. Then the following holds:

$Pon = MIN_{0 \leq i \leq n-1} Pon_i$

Assume that component C_i has left context L_i and right context R_i and that the whole structure has left context L and right context R. Then the following holds:

$L = L_i$, for $0 \leq i \leq n-1$

$R = R_i$, for $0 \leq i \leq n-1$

Frame 2. Description of a P structure.

APPOG, a collateral successive structure

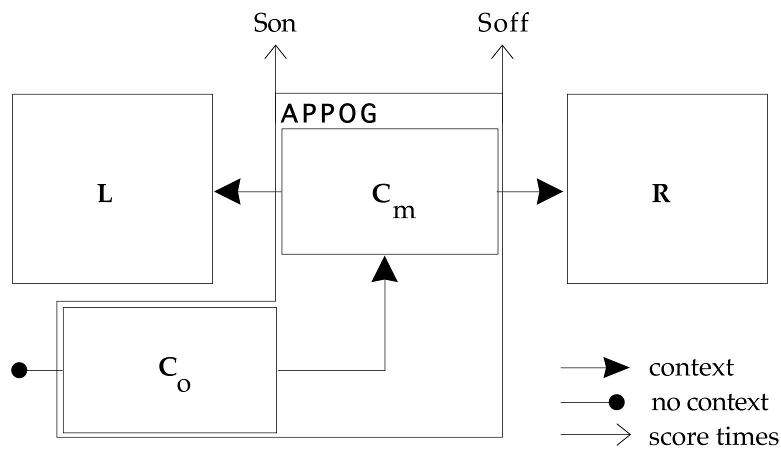

*Figure of **APPOG** object*

Consider a APPOG structure of a ornament component C_o and a main component C_m. Assume that component C_o has score onset time Son_o, score offset time $Soff_o$, that component C_m has score onset time Son_m and score offset time $Soff_m$ and that the whole structure has score onset time Son and score offset time Soff. Then the following must hold:

$Son_m = Son$
$Soff_m = Soff$
$Soff_o = Son_m$

Assume that component C_o has performance onset time Pon_o, component C_m has performance onset time Pon_m and that the whole structure has performance onset time Pon. Then the following holds:

$Pon = Pon_m$
$Pon_o \leq Pon_m$

Assume that component C_o has left context L_o and right context R_o, component C_m has left context L_m and right context R_m and that the whole structure has left context L and right context R. Then the following holds:

$L = L_m$
$R = R_m$
$R_o = C_m$
$L_o =$ *undefined*

Frame 3. Description of an APPOG structure.

ACCIA, a collateral simultaneous structure

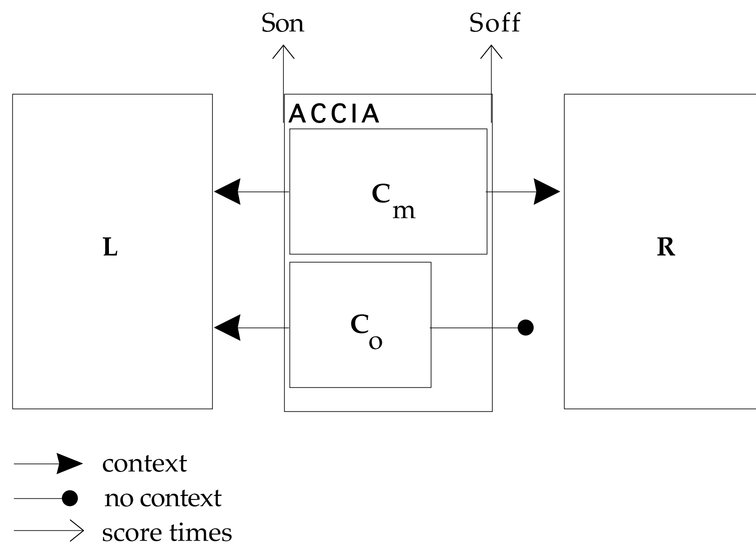

*Figure of **ACCIA** object*

Consider a ACCIA structure of a ornament component C_o and a main component C_m. Assume that component C_o has score onset time Son_o, score offset time $Soff_o$, that component C_m has score onset time Son_m and score offset time $Soff_m$ and that the whole structure has score onset time Son and score offset time Soff. Then the following must hold:

$Son_o = Son_m = Son$
$Soff_m = Soff$

Assume that component C_o has performance onset time Pon_o, component C_m has performance onset time Pon_m and that the whole structure has performance onset time Pon. Then the following holds:

$Pon = Pon_m$

Assume that component C_o has left context L_o and right context R_o, component C_m has left context L_m and right context R_m and that the whole structure has left context L and right context R. Then the following holds:

$L = L_m = L_o$
$R = R_m$
$R_o =$ *undefined*

Frame 4. Description of an ACCIA structure.

EXAMPLE OF THE REPRESENTATION OF A MUSICAL OBJECT

In figure 2 a fragment of a score is shown that will serve as a basis for the examples at the end of this article. It is the score of the last bars of the theme of six variations over the duet *Nel cor più non mi sento*, by Ludwig van Beethoven (with some adaptations), which is the same material used to study tempo curves in (Desain & Honing, 1992a). It contains examples of several kinds of musical structure: chords, voices, sequences, bars and beats, phrases and two types of ornaments. Figure 3 shows a graphical notation indicating two structural descriptions: a metrical hierarchy and a separation into voices. The way these structures are specified in Lisp is given in the appendix.

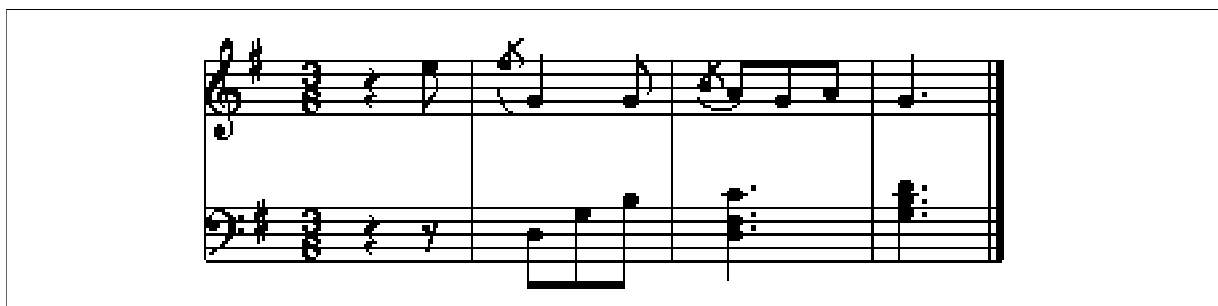

Figure 2. Score of the last bars of the theme of six variations over the duet Nel cor più non mi sento, *by Ludwig van Beethoven.*

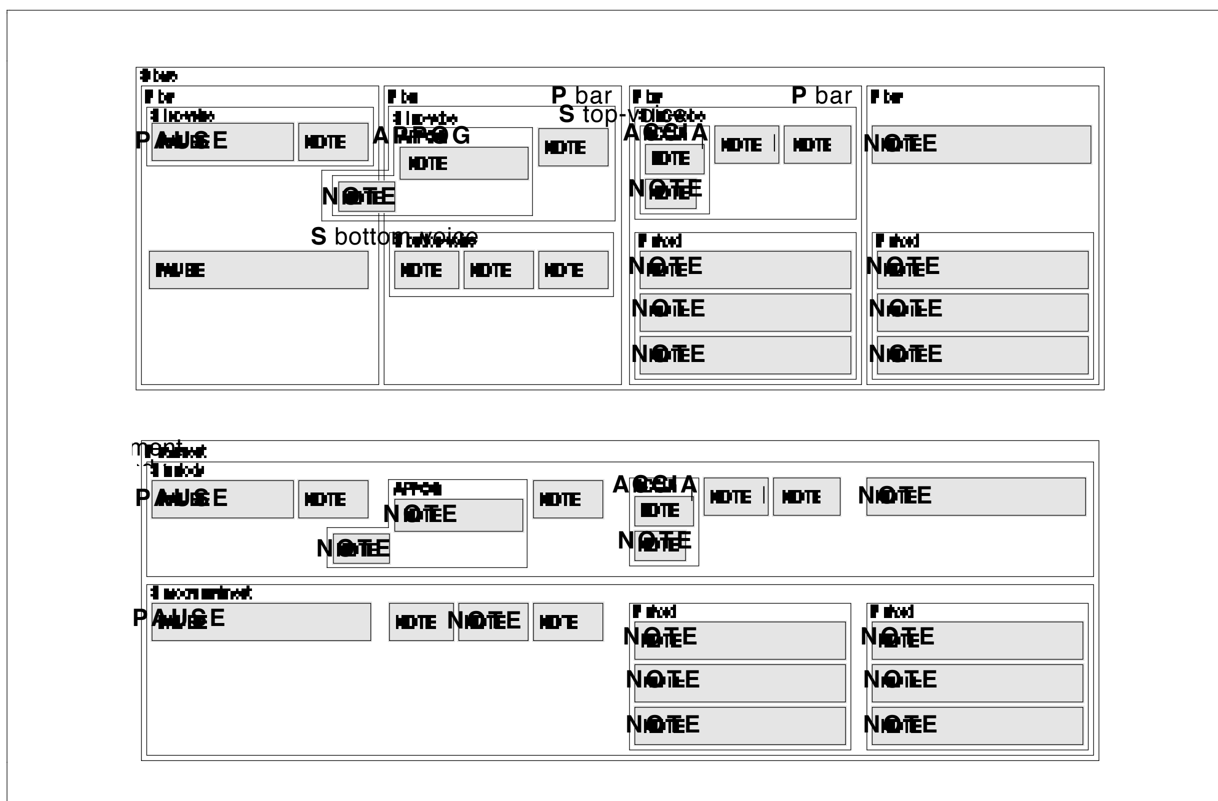

Figure 3. a) Structural description of the metrical hierarchy of the score in figure 2, and b) Structural description of the voices in that piece.

REPRESENTING EXPRESSION

There are three kinds of expressive timing: expressive tempo, expressive asynchrony and expressive articulation. The first two are based on performance onset times only, the third is based on performance onset and offset times (see figure 4).

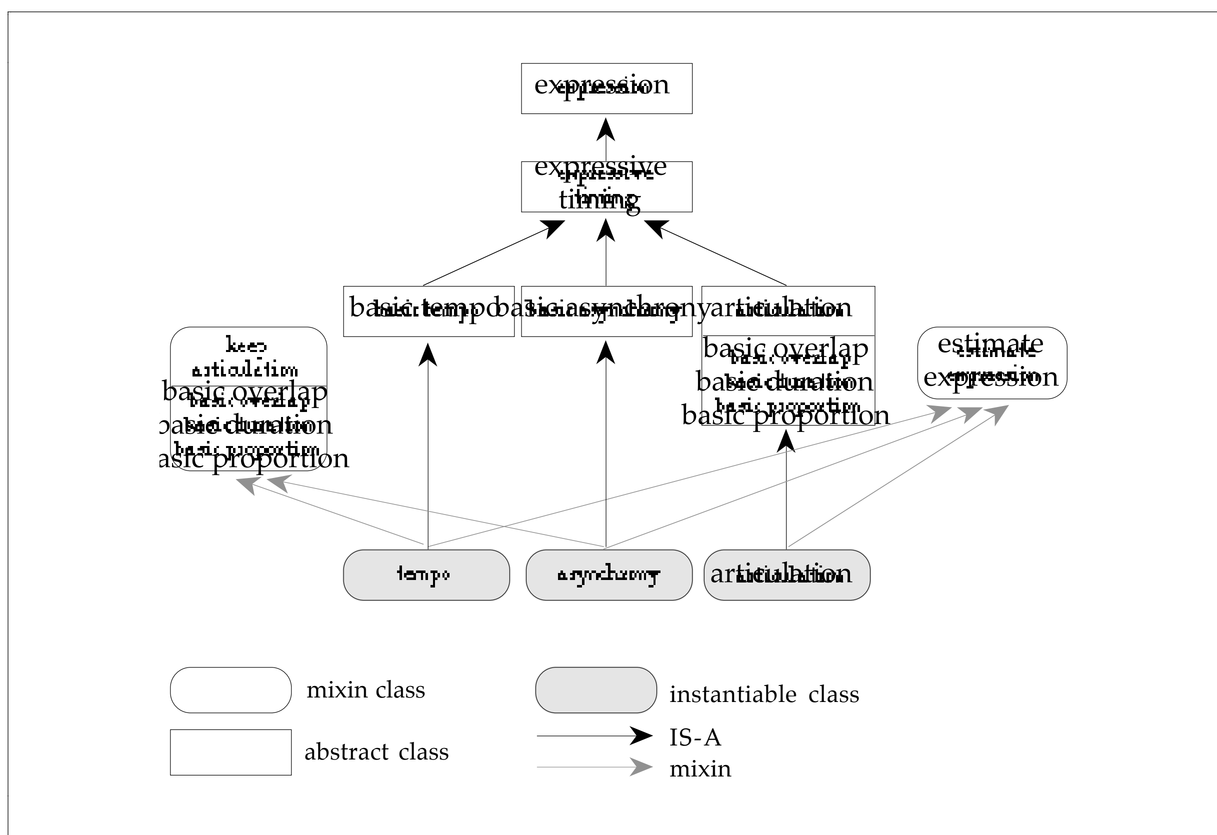

Figure 4. Expression type hierarchy.

One could imagine sophisticated algorithms that calculate the onset of a note and of parallel structures on the basis of their perceptual onset (P-center; see Vos & Rasch, 1981). But for clarity we use a very simple definition of onset times, which was already given in the frames 1 to 4. In that way, all musical objects have performance onset times and so can be used as units on which tempo and asynchrony measures are built.

Expressive tempo

The notion of tempo is relevant only for successive structures. It is defined as the ratio of score duration and performance duration. This velocity-like notion the inverse of the notion of a tempo factor, as is used in the psychology of music literature.

Expressive asynchrony

The notion of asynchrony is relevant only for simultaneous structures. It is defined as the difference of performance onsets. It is thus independent of score times.

Expressive articulation

Expressive articulation uses the performance offsets of individual notes. It simply assumes that they are given. A definition of performance offset of structured musical objects is not needed. Articulation is also independent of score times.

Articulation can be defined in several ways - but it is hard to find a way that will suffice in all circumstances. In the legato range the absolute overlap time of the sounding part of a note and the next one seems a good candidate for an articulation scale. In the staccato range the absolute sounding duration of the note seems the most prominent perceived aspect. In the intermediate range the relative sounding proportion is a good measure. For the moment we cannot do better than to supply these three concepts of articulation expression (overlap-, duration- and proportion-articulation) - leaving it for the user to choose the most appropriate one (see frame 5). For a multilateral structure the expressive articulation value is taken to be the average articulation of its parts. For a collateral structure the expressive articulation value is defined to be the articulation of its main part.

Definition of articulation

Consider a note with performance onset Pon, performance onset Poff and performance onset of its right context Pon_r. There are three alternative definitions of articulation A given:

overlap articulation $A = Poff - Pon_r$

duration articulation $A = Poff - Pon$

proportion articulation $A = \frac{Poff - Pon}{Pon_r - Pon}$

If a multilateral structure with articulation A has components C_i for $0 \leq i \leq n-1$, and C_i has articulation A_i then:

$$A = \mathrm{MEAN}_{0 \leq i \leq n-1} A_i$$

If a collateral structure has articulation A, and its main component C_m has articulation A_m then:

$$A = A_m$$

Frame 5. Definition of articulation expression.

Estimate onsets

Because sometimes the performance onset of missing objects (like the virtual note after the end of the piece) or the performance onset of a rest are needed, we devised a set of procedures that estimates these missing values on the basis of performance onsets that can be found in the context, using a linear interpolation or extrapolation method. The set of procedures forms a mix-in class that can be combined with any expressive timing type enabling that kind of expressive timing to deal - in all operations - with missing values. Estimation is derived from the same structural level as the transformation itself. For example, a transformation on a beat structure in need of a missing expressive value at the end of the piece (cf. the onset the final barline in a score) will be estimated on basis of the two previous beats -not on the basis of any internal detail. In the case of extreme tempo variations, as occur in a final retard, the estimation feature cannot work well. In this case it is better not to use it.

Articulation invariance

When moving the onsets of notes around (e.g. in modifying the performance onsets) it is quite annoying that the articulation of the individual notes also changes - an effect that is very easy to perceive and which may well overshadow subtle modifications of onset timing. Therefore a set of procedures can be mixed-in with expressive tempo and asynchrony. They are given a chance to calculate the articulation of individual notes before onsets are changed and to reinstall it afterwards. This will insure that articulation is kept invariant under transformations of onset timing (see figure 12).

EXPRESSION MAPS

An abstraction of the expression of an object is useful for many operations because it can hide the irrelevant details of the structure and provides a means to transfer expression from one object to another. Therefore expression maps were introduced. They describe expression of musical objects at one level of a structural description. All objects at the level described must have the same structural type. Maps contain a list of sections, one for each of those musical objects. A section lists the expressive values of the components of that musical object. Of course maps may be partial - consisting of several sections with gaps in between, or even have missing values within a section.

Onset timing

The application of a (modified) map of performance onsets on an object works as follows. First, all objects at the indicated level are found, paired with their corresponding sections. Then each section is applied to its object. This means that the components of that object are provided one by one with a new onset from that section.

This setting of onsets is handled differently according on the structural type of the component. If this component is a note, the onset is set directly. For S components the whole structure is stretched between that onset and and the next onset (the onset of the succeeding component). A P component is set to the provided onset, but keeps its internal asynchrony invariant and truncates at the next onset. In the case of a ACCIA component, the main structure is set to the onset, with the ornament following the displacement of the main structure. Finally, for a APPOG component, the main structure is stretched between that onset and the next onset, with the ornament also simply following the displacement of the main object.

Now we have indicated how an expressive timing is applied to components of structured objects - it remains to be shown how such a change propagates when these components again are embedded structured objects themselves. This fairly complex process depends on the type of the embedded structured object and mirrors the decisions given above: S components are stretched, P components are shifted and truncated, and ornaments follow the shift of their main components.

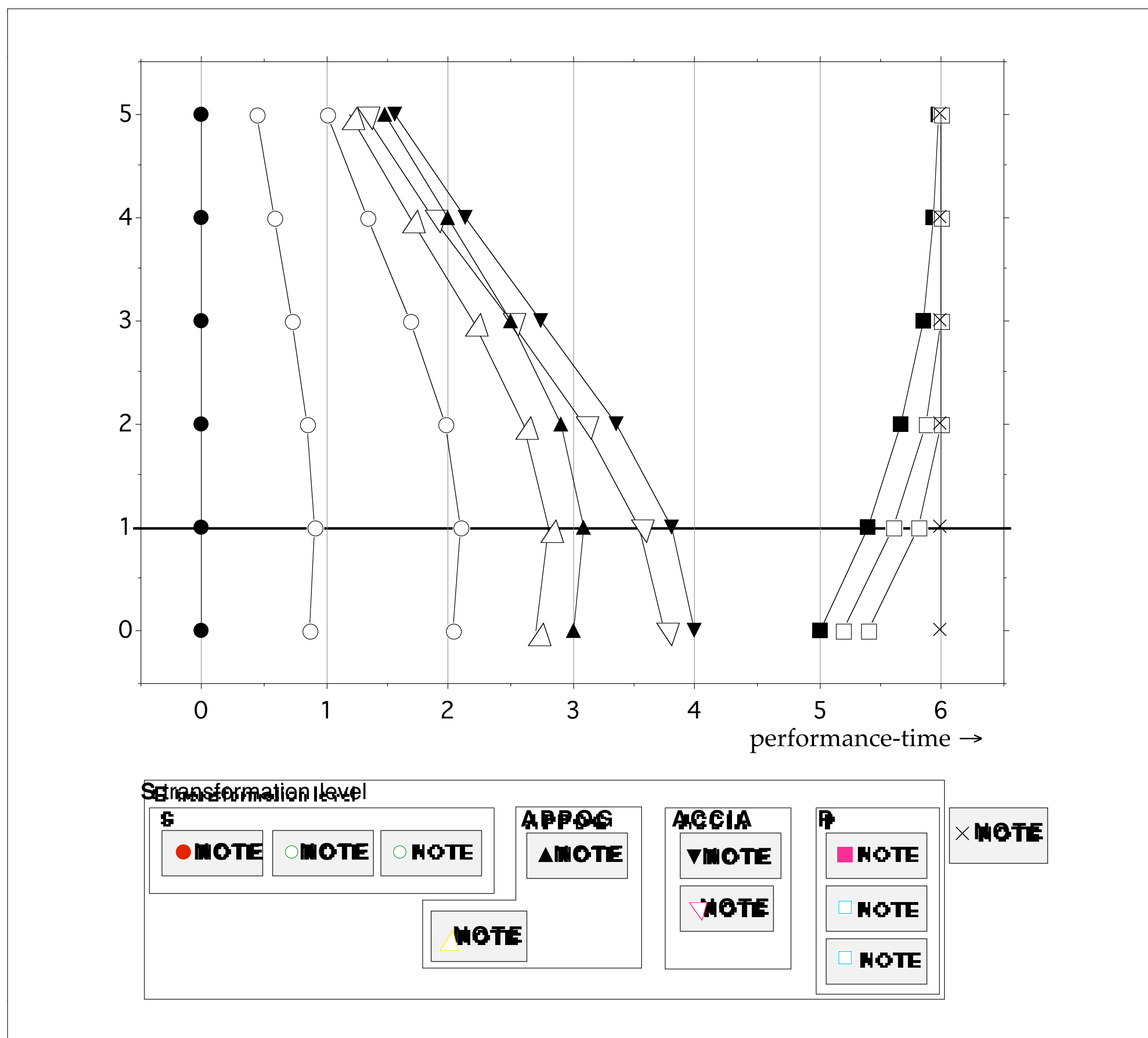

Figure 5. Propagation of change of onset within an S structure for different component types. This figure shows the propagation process for an S structure containing different types of structural components. We assume the components are moved around by an arbitrary transformation, parametrized by a factor. In this figure it is shown how this change is propagated to the internal structure of different kinds of components. The first component is an S structure and the onsets of its internal parts (lines marked with white circles) are stretched along proportionally. The second sub-structure is an APPOG structure and one can see that the onset of its ornament (line marked with upward pointing white triangles) shifts along with the main object. The third sub-structure is an ACCIA structure and the onset of its ornament behaves likewise. Note that the onset of the ornament is allowed to shift freely (line marked with downward pointing white triangles), even the order of notes is allowed to change here . The fourth sub-structure is a P structure and the onset of its components (lines marked with squares) are shifted and truncated at the end (the right context note; line marked with x's).

Articulation expression

In comparison, to set the articulation expression to a structured object is much simpler. When a section of an articulation map is applied to a multilateral or collateral structure the articulation of its components are set to their respective values from the section.

The propagation of a (modified) articulation value to a component works as follows. If that component is a note, a new offset is calculated from the articulation value and set directly, taking care to maintain reasonable offset times (e.g. not shifting before its onset). If that component is a multilateral structure, its articulation is calculated (the mean articulation of its components) and the difference with the required articulation is propagated as an increment to all components. If it is a collateral structure, its

articulation is calculated (the articulation of its main component) and the difference with the required articulation is propagated as an increment to both main and ornament components.

OPERATIONS ON EXPRESSION MAPS

Operations on expression maps work section by section. In each section the expression of a structured musical object is represented. The operations delivers a new section to be applied to that object. Care was taken to maintain structural consistency in all operations even in case of extreme parameter values. Of course expression transformations are intended as subtle changes and truncation or extreme normalization should in practice never occur.

Scale maps

Scaling expressive tempo

Scaling tempo is done in an exponential way. Inverse tempi are considered to be related by a scale factor -1; twice as slow is considered to be the mirror image of half as slow. This exponential scaling of expressive tempo mirrors the exponential nature of notated note durations.

Scaling the expressive tempo of an S section

The scaling of the expressive tempo of a multilateral successive structure works as follows. Assume the structure has n components named C_i with $0 \leq i \leq n-1$. Assume component C_i has score onset time Son_i and performance onset time Pon_i . Assume the right context of the structure (and thus the right context of component C_{n-1}) is object C_n . It has score onset Son_n and performance onset time Pon_n . A section of the expressive tempo map of the structure contains all Son_i and Pon_i including Son_n and Pon_n . The scale operation on such a section delivers a new section with performance onsets $Pon_i{}'$ according to the following rules:

Define the score inter-onset interval ΔSon_i and the performance inter-onset interval ΔPon_i and the local tempo T_i for $0 \leq i \leq n-1$ (a better term would be velocity) as:

$$\Delta Son_i = Son_{i+1} - Son_i$$
$$\Delta Pon_i = Pon_{i+1} - Pon_i$$
$$T_i = \frac{\Delta Son_i}{\Delta Pon_i}$$

This ratio is scaled by an exponential factor f.

$$T_i{}' = T_i{}^f$$

Then new raw performance durations $\Delta Pon_i{}''$ are calculated:

$$\Delta Pon_i{}'' = \frac{\Delta Son_i}{T_i{}'}$$

These are re-normalised such that the total performance duration is kept invariant.

$$\Delta Pon_i{}' = \Delta Pon_i{}'' * \frac{Pon_n - Pon_0}{\sum_{i=0}^{n-1} \Delta Pon_i{}''}$$

Starting at the same point, the new performance times are given as:

$$Pon_i{}' = Pon_0 + \sum_{j=0}^{i-1} \Delta Pon_j{}'$$

Frame 6. Scaling the expressive tempo an S section.

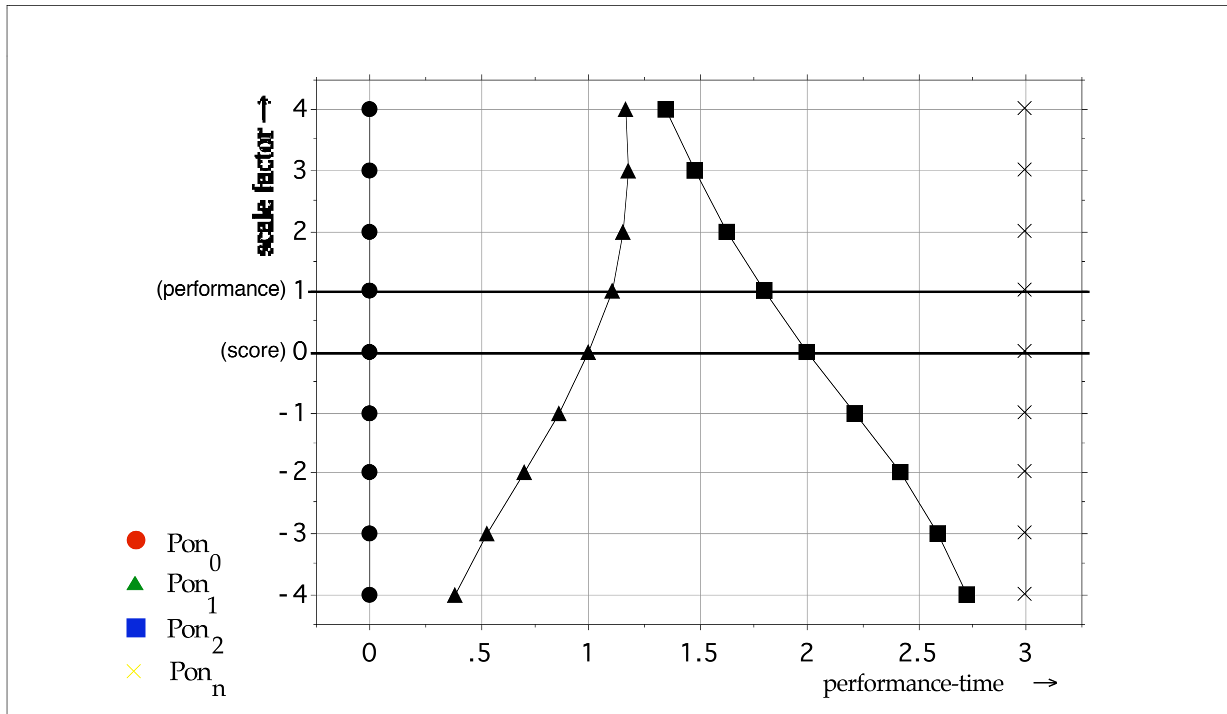

Figure 6. Scaling the expressive tempo of an S section. This process is shown for a specific set of performance onsets Pon_i .In this figure the horizontal axis is the performance time P. On the vertical axis the scale factor f is given. Thus at the horizontal line at scale factor 1 the performance times Pon_i ' are shown as markers on the line; they are identical to the original performance times Pon_i . This operation (with scale factor 1) is the identity transformation with respect to the performance timing. At the horizontal line at scale factor 0 the performance times Pon_i ' are identical to the score times Son_i (modulo normalization to the total performance duration) . This operation (with scale factor 0) effectively removes the expressive timing of the performance. At factor .5 a diminished expressive timing profile will result, and at factor 2 an exaggerated rubato can be obtained. At negative values of the scale factor the expressive profile is inverted: a slower tempo becomes faster and vice versa. At extreme values of the scale factor the note that is played at the slowest tempo in the performance will gain almost the whole performance time interval spanned by the structure, pushing other notes to zero duration.

When the performance onset Pon_n is not available, the scale transformation uses Pon_{n-1} instead, and scales the tempo of the section with regard to the onset of the last component in the section - instead of the onset of the right context. This tempo scaling method works well for S constructs with many components and small tempo deviations.

Scaling the expressive tempo of an APPOG section

The scaling of the expressive tempo of a collateral successive structure works as follows. Assume this structure has a main component with score onset time Son_m and performance onset time Pon_m and a preceding ornament component with score onset time Son_o , and performance onset time Pon_o . Assume the right context of the structure (and thus the right context of component C_m) is object C_r . It has score onset Son_r and performance onset time Pon_r . An APPOG time map section contains this score and performance data. The scale operation on such a map delivers a new map with performance onsets according to the following rules:

Define the main and ornament score inter-onset interval ΔSon_m , ΔSon_o and the main and ornament performance inter-onset interval ΔPon_m , ΔPon_o as:

$$\Delta Son_m = Son_r - Son_m$$
$$\Delta Son_o = Son_m - Son_o$$
$$\Delta Pon_m = Pon_r - Pon_m$$
$$\Delta Pon_o = Pon_m - Pon_o$$

The ornament tempo T_o and the main tempo T_m are calculated as:

$$T_o = \frac{\Delta Son_o}{\Delta Pon_o}$$
$$T_m = \frac{\Delta Son_m}{\Delta Pon_m}$$

$T_{o/m}$ is tempo of the ornament relative to the main tempo. This factor is scaled by an exponential parameter f, and a new ornament tempo T_o' is calculated:

$$T_{o/m} = \frac{T_o}{T_m}$$
$$T_o' = T_m * T_{o/m}^{f}$$

This gives a new performance duration $\Delta P_o'$, which yields the new performance times Pon_m' and Pon_o':

$$\Delta Pon_o' = \frac{\Delta Son_o}{T_o'}$$
$$Pon_m' = Pon_m$$
$$Pon_o' = Pon_m - \Delta Pon_o'$$

Frame 7. Scaling the expressive tempo of an APPOG section.

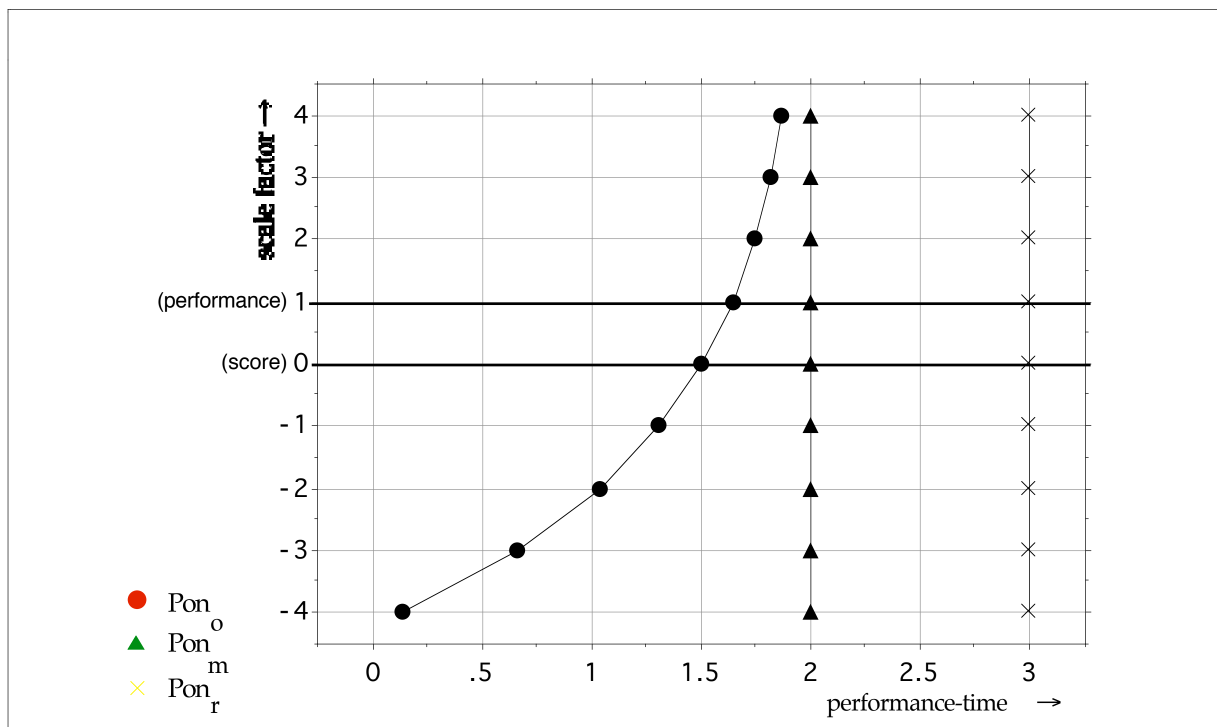

Figure 7. Scaling expressive timing of an APPOG section. This process is shown for a specific set of performance onsets. Note that only the performance timing of the ornament is affected. At scale factor 1 the timing of the ornament is identical to the original timing. At scale factor 0 the ornament is performed at the same tempo as the main object (in this particular example the score duration of the ornament is half that of the main component). This operation (with scale factor 0) effectively removes the expressive way in which the ornament is performed, relative to the main component. At factor .5 a diminished expressive timing effect will result, and at factor 2 an exaggerated effect will be obtained. At negative values of the scale factor the expressive timing is inverted: a performance of the ornament at a lower tempo than the main component becomes one at a faster tempo and vice versa.

Scaling expressive asynchrony

Asynchrony occurs when two or more simultaneous musical objects - prescribed to happen at the same score time - have unequal performance onsets. The differences can be scaled linearly but care has to be taken not to disrupt the timing of higher levels.

Scaling the expressive asynchrony of a P section

The scaling of the expressive asynchrony of a multilateral simultaneous structure works as follows. Assume the structure has n components named C_i with $0 \le i \le n-1$. Component C_i has performance onset time Pon_i. Assume the right context of the structure (and thus the right context of all components) has performance onset Pon_n. A parallel time map of the structure contains all Pon_i including Pon_n. The scale operation on such a map delivers a new Pon_i' according to the following rules:

Let the global performance onset Pon and the performance onset asynchronies ΔPon_i be defined as:

$$Pon = MIN_{0 \le i \le n-1} Pon_i$$

$$\Delta Pon_i = Pon_i - Pon \text{ for } 0 \le i \le n-1$$

The asynchronies are scaled by an multiplication factor f:

$$\Delta Pon_i' = \Delta Pon_i * f$$

New performance onsets Pon_i' are calculated, shifting such that the global performance onset is kept invariant ($\min(Pon_i') = \min(Pon_i) = Pon$). The result is truncated such that the onsets never move beyond Pon_n. Of these two safeguards the first applying in case f is negative, the second applying in case f is large compared to the ratio of the asynchronies and performance duration of the whole structure. Together they ensure consistency with higher-level structural descriptions by keeping the components within the bounds of the structure.

$$Pon_i' = MIN(Pon_n, Pon + \Delta Pon_i' + MIN(\Delta Pon_i'))$$

Frame 8. Scaling the expressive asynchrony of a P section

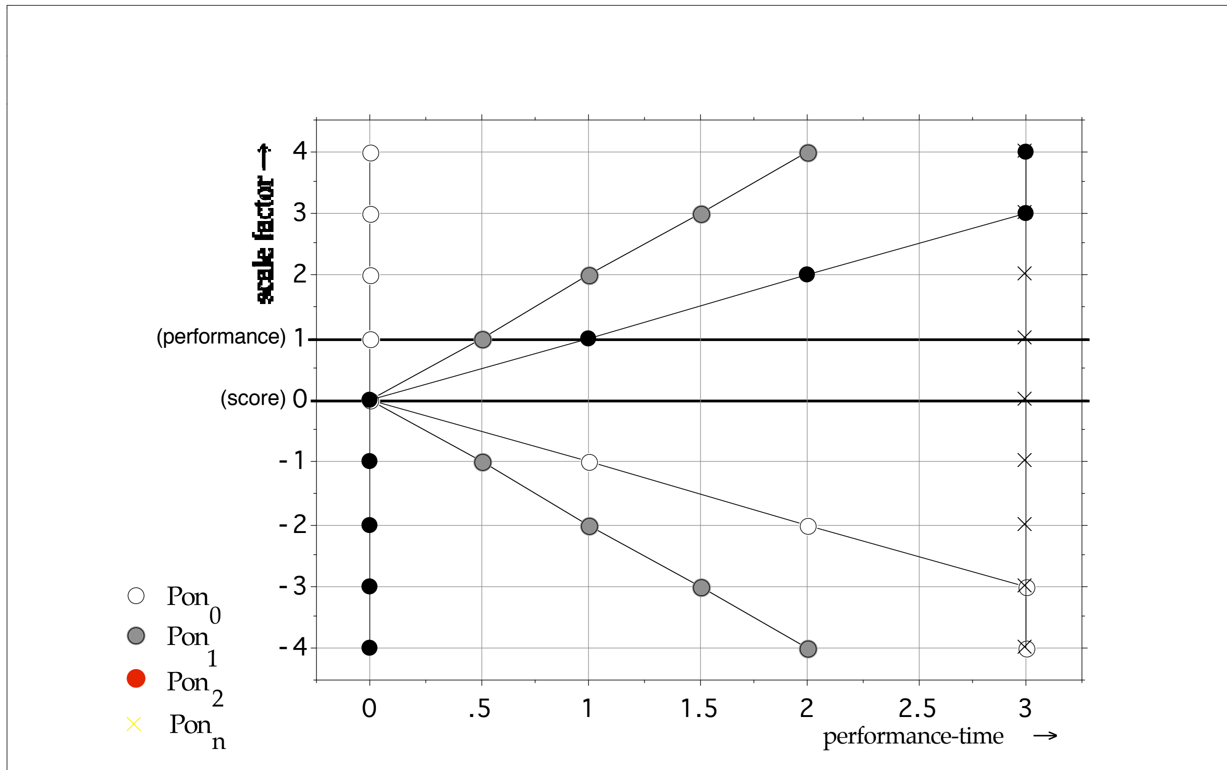

Figure 8. Scaling expressive timing of a P section. This figure shows this process for a specific set of performance times P_i (say a chord performed with some spread) . At scale factor 1 the performance onsets Pon_i ' are identical to their original Pon_i . At scale factor 0 all Pon_i ' occur synchronously at the minimum of their originals (i.e. removed chord spread). At factor .5 a diminished chord spread will result, and at factor 2 an exaggerated chord spread can be obtained. At negative values of this factor the spread is inverted: first notes becoming last and vice versa. At extreme values of the scale factor the notes are restrained from moving out of the chord structure into the next musical object by truncation. Note that the whole operation is independent of score times.

Scaling the expressive asynchrony of an ACCIA section

The scaling of the expressive asynchrony of a collateral simultaneous structure works as follows. Assume the structure has a main component with performance onset time Pon_m and an ornament component with performance onset time Pon_o . A time-map of the structure contains Pon_o and Pon_m . The scale operation on such a map delivers new performance onsets according to the following rules:

Let the performance onset asynchrony ΔPon be defined as:

$$\Delta Pon = Pon_o - Pon_m$$

The asynchrony is scaled by a multiplication factor f, and a new performance onset Pon_o' is calculated:

$$\Delta Pon' = \Delta Pon * f$$
$$Pon_o' = Pon_m + \Delta Pon'$$
$$Pon_m' = Pon_m$$

Frame 9. Scaling the expressive asynchrony of an ACCIA section.

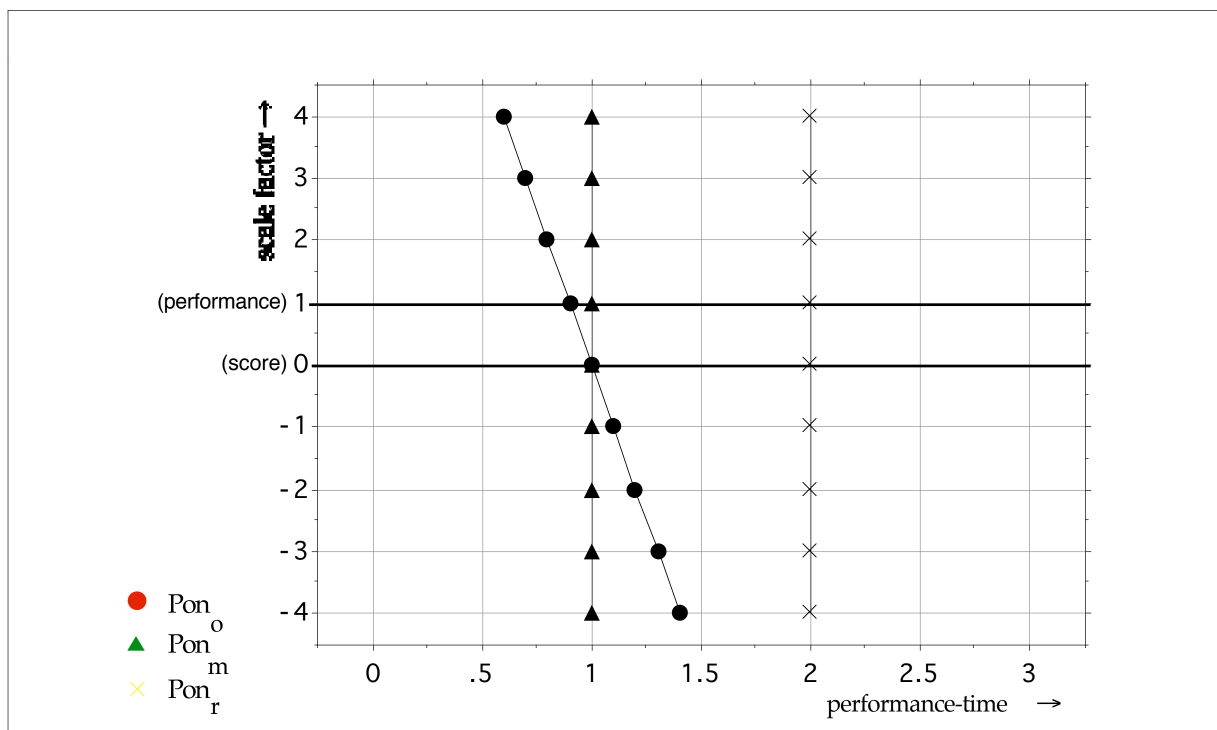

Figure 9. Scaling expressive timing of a ACCIA section. It shows this process for a specific set of performance times (a note preceded by an acciaccatura). At scale factor 1 all performance onsets are identical to their original. At scale factor 0 the ornament occurs synchronously with the main note (removed asynchrony). At negative values of this factor the order of onset of ornament and main note is inverted. Note that the ornament is allowed to shift freely - even outside the bounds of the whole ACCIA structure.

Scaling expressive articulation

The articulation of a note is interpreted (scaled) relative to the articulation of the structure that it forms part of. For multilateral structures this is the average articulation. If thus the first note in a bar is played with more overlap than the other notes, a removal of the overlap articulation expression (a zero scale factor) will set the overlap of all notes to the mean overlap of the notes in the structure. And exaggerating the articulation expression (a scale factor larger than 1) will move the individual overlaps away from the mean - but maintaining the average overlap of all the notes in the bar. Of course all articulation types maintain reasonable performance offsets in the case of extreme values (i.e. note offsets will not shift before their onsets).

Scaling the expressive articulation of a multilateral section

Assume a multilateral structure has n components C_i with $0 \leq i \leq n-1$. Component C_i has articulation A_i (see frame 5 for the calculation of A_i). A section of the expression map of the structure contains all A_i. The articulation A of the structure itself is defined as:

$$A = MEAN_{0 \leq i \leq n-1} A_i$$

Let the expression deviations be

$$\Delta A_i = A_i - A \text{ for } 0 \leq i \leq n-1$$

The deviations are simply scaled by a multiplication factor f

$$\Delta A_i' = f * \Delta A_i$$

The scale transformation delivers new articulations A_i' by adding the new deviations to the reference articulation A such that the articulation of the whole structure is kept invariant (mean A_i' = A).

$$A_i' = A + \Delta A_i'$$

Keeping the expressive values in a reasonable range can only be done while applying them to the individual notes.

Frame 10. Scaling the expressive articulation of a multilateral section

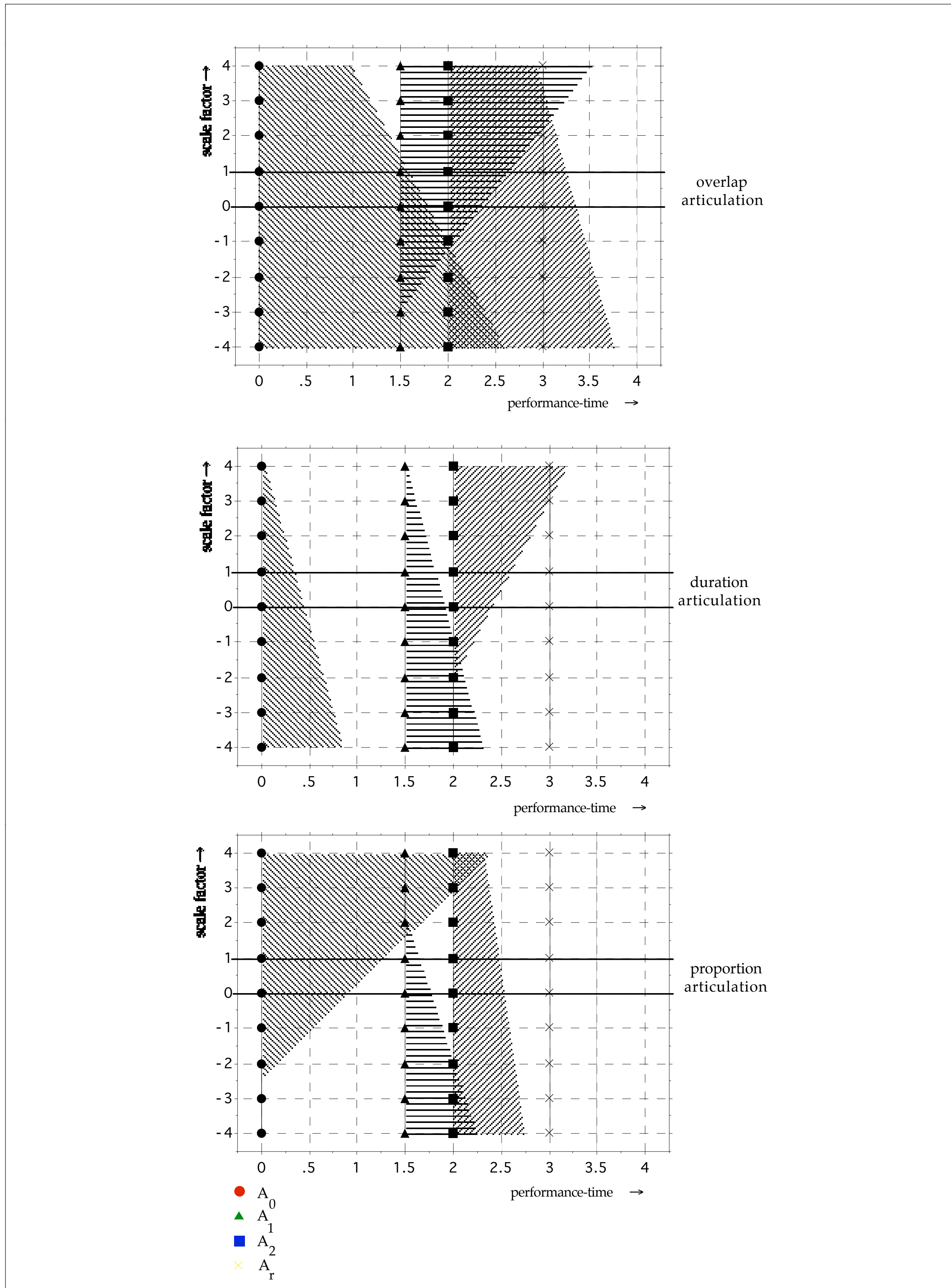

Figure 10. Scaling of an S section with three different kinds of articulation. It shows the scaling of three types of articulation for a multilateral structure, in this instance an S structure with a specific set of performance onset and offset times. Here, at scale factor 1 articulations A_i' are identical to the original performance. At scale factor 0 all A_i' are scaled to the mean articulation A. At a scale factor above 1 the deviation of each A_i' with respect to A is exaggerated, with negative values constituting an inverse deviation: legato notes become more staccato and vice versa. Note that the mean articulation A is always kept invariant.

Scaling the expressive articulation of a collateral section

Assume that a collateral structure has ornament and main components C_o and C_m. Component C_o has articulation A_o and component C_m has articulation A_m (see frame 5 for the calculation of A_o and A_m). A section of the expression map of the structure contains these values. The articulation A of the structure itself is defined as:

$A = A_m$

Let the expression deviation be

$\Delta A = A_o - A$

The deviation is scaled by a multiplication factor f

$\Delta A' = f * \Delta A$

The scale transformation delivers a new articulation for the ornament by adding the new deviation to the reference articulation A.

$A_o' = A + \Delta A'$
$A_m' = A_m$

Keeping the expressive values in a reasonable range can only be done while applying them to the individual notes.

Frame 11. Scaling the expressive articulation of a collateral section.

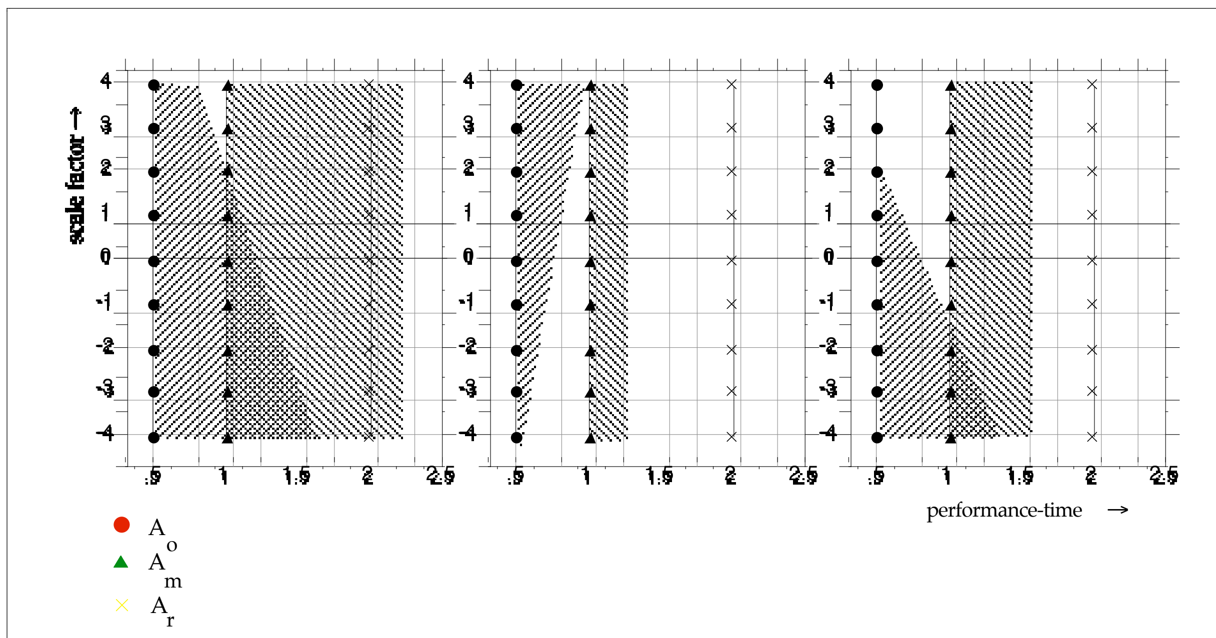

Figure 11. Scaling of an APPOG section with three different kinds of articulation. It shows three types of articulation scaling for an ornament (here an APPOG structure). At scale factor 1 the articulation A_o' is identical to the original articulation of the ornament. At scale factor 0 A_o is identical to the articulation of the main component A_m At a scale factor above 1 the deviation of A_o with respect to the main component A_m is exaggerated, negative values constituting an inverse articulation: legato ornament articulation become more staccato and vice versa.

Keeping articulation consistent in the scaling of expressive timing

In the scaling of timing of onsets we ignored the influence it should have on its offsets. To obtain some sort of articulation consistency we can use the three types of articulation (as described above) when scaling expressive tempo and expressive asynchrony. In figure 12, we use expressive tempo scaling for an S section as an example in illustrating the different types of articulation consistency.

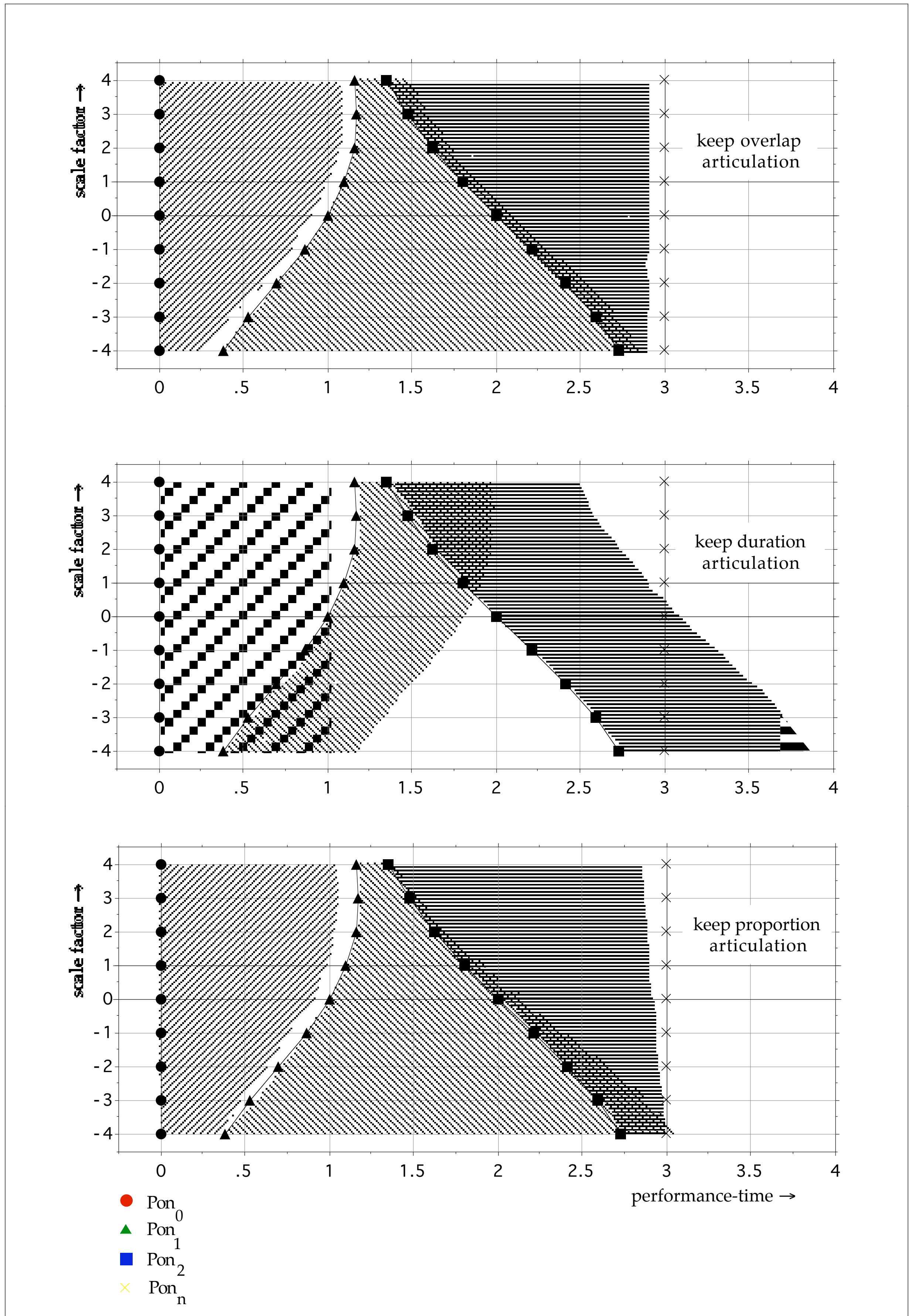

Figure 12. Scaling of an S section that keeps a particular type of articulation consistent. Shown for the same set of performance onsets as used in figure 6.

Stretch maps

Sometimes it is useful to be able to keep a consistency in performance timing between voices when modifying one of them. Naming the modified material as the foreground and describing the rest as the background, the consistency requires that a series of performance onsets, at a selected background level, that happen between two performance onsets in the foreground are "stretched along" with the changes in the foreground. This feature is implemented by first extracting a timing map from the background, and "stretching" this map between the old and modified foreground map before it is reapplied to the background. The fore- and background must be parallel (must happen during the same score time interval) and have to be S structures. Maintaining the consistency between other kinds of structure remains a problem.

Interpolate maps

A more sophisticated notion of expression entails the difference in expression between two structured objects. The best known example is voice leading in ensemble playing (Rasch, 1979) whereby the leading instrument often takes a small but consistent timing lead (around 10 ms). Inter- or extrapolation between two extracted timing maps yields the possibility to scale this kind of expression.

Transfer maps

Sometimes it is useful to apply an expression map extracted from one object, to another object, possibly with a different structure, e.g. boldly applying the expressive timing of the melody to the accompaniment. This is supported via an operation on timing maps that uses the structure of one map but imposes expressive values of the other.

TRANSFORMATIONS

Transformations of musical structures are generalizations of the operations on expression maps. They handle the selection of a level of structural description, extract a map, do the operation and re-impose the map. However, they often become quite sophisticated because they also take care of maintaining consistency with a background (material that is not affected directly). The application of the modified map has its own complexity, whereby changes are propagated to lower levels depending on the types of musical structure encountered. Finally, in the setting of new performance onsets of the notes, also the offsets may change in order to keep the articulation invariant. Out of the wealth of possibilities we choose some examples to be illustrated further by means of figures. In the figures the performance onsets and/or offsets of the individual notes at different parameter settings are given. The structure of the musical objects transformed are shown underneath.

In the following examples the same performance of a Beethoven theme is used (the fragment as shown in figure 2), allowing for comparison of the different transformations and to see the effect of applying the same transformation to different levels or types of structure. Note that for all the transformations the indentity transformation is shown at scale factor 1.

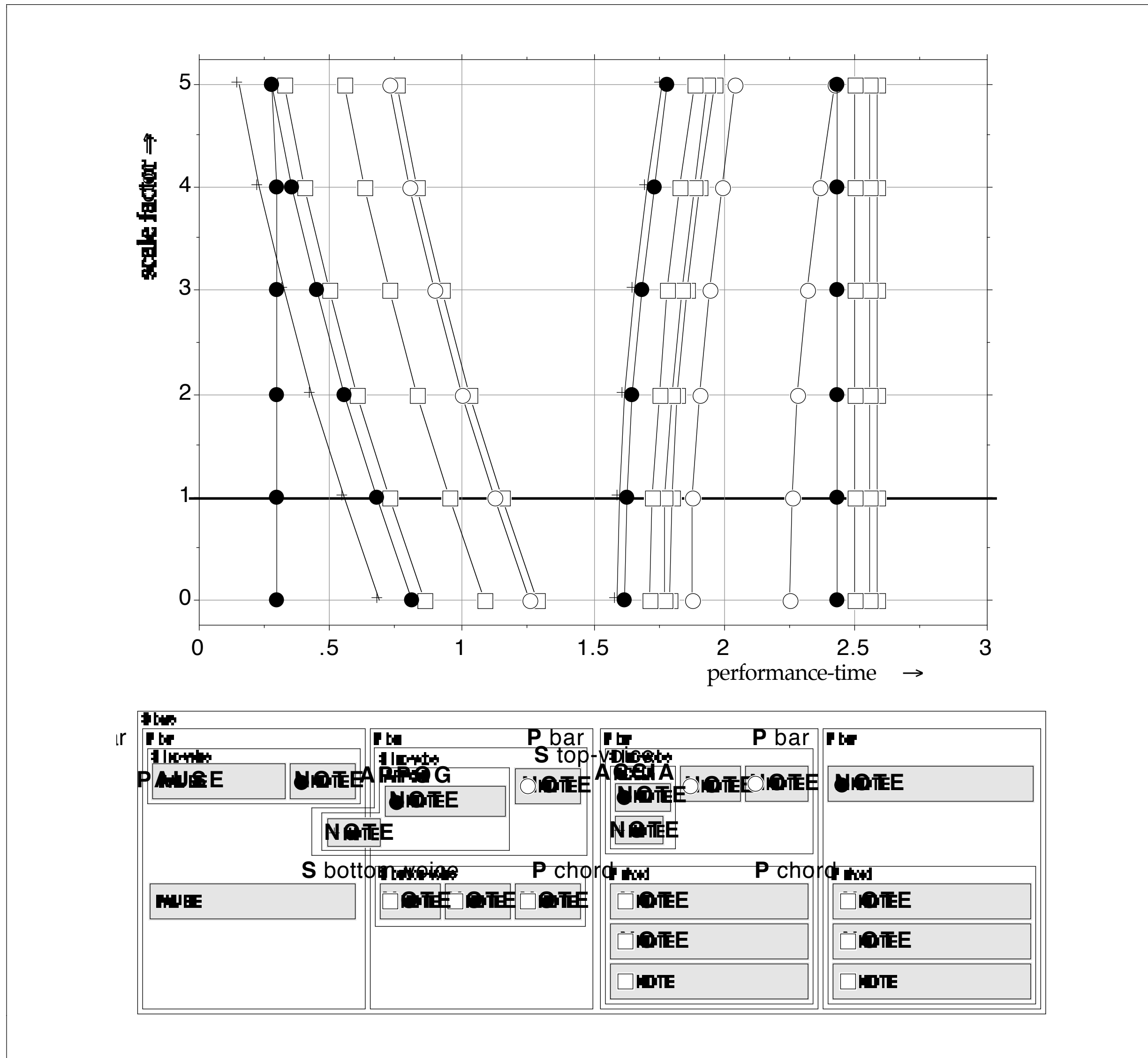

Figure 13. Scaling the expressive tempo of bars in the Beethoven fragment. Underneath the figure a structural description of the fragment is shown in bars. Imagine what would happen if we asked a performer to emphasize his/her timing of the bars? One possibility would be to play the onsets of the bars, that were played slightly early, even earlier, and ones that were played late, later still. This particular transformation can be read from figure 12 as the lines with the black markers, indicating the component in the bar that carries the expressive timing. Both the performance onset of the first and the last bar of the enclosing bars' structure are not changed; the transformation is done at the level named "bars", with its timing kept invariant. The lines with white markers show the embedded material that follows the change of the performance onset of each bar. Note that the timing of the ornamented notes does not change (they keep the same distance with respect to the note they cling to), as does the spread of the chords.

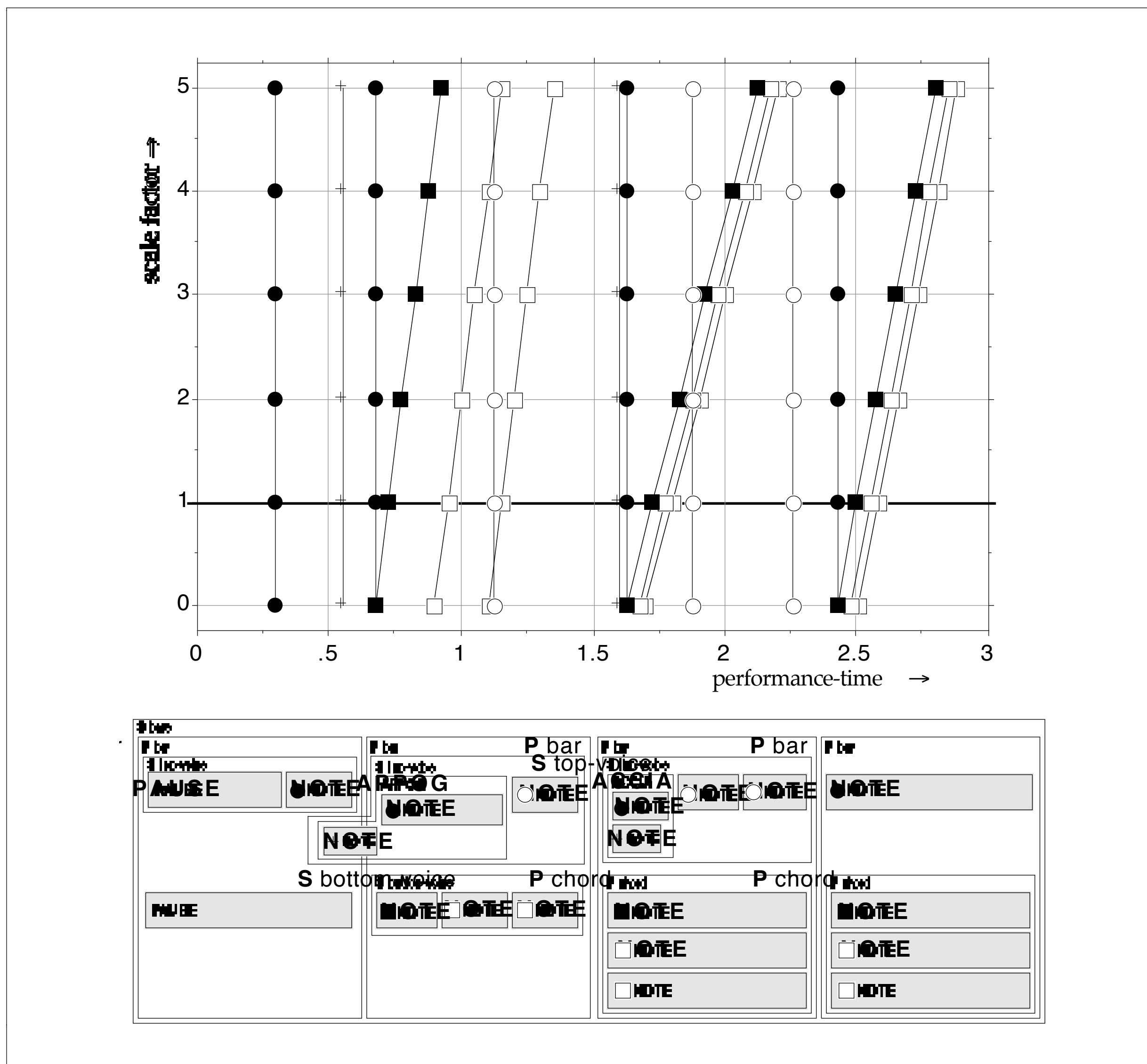

Figure 14. Scaling the expressive asynchrony of each bar in the Beethoven fragment. It shows the expressive transformation we might expect to happen when a performer is asked to exaggerate the asynchrony between the top-voice and bottom-voice at the onset of each bar. The figure shows the scaling of the asynchrony of the bottom voice onsets (the black squares), without changing the timing of the bars (lines marked with black circles and triangles). The embedded notes of the bottom voice (lines with white squares) just shift along with the expressive timing of their embedding structure. Here again, the ornament timing and the chord spread stay invariant.

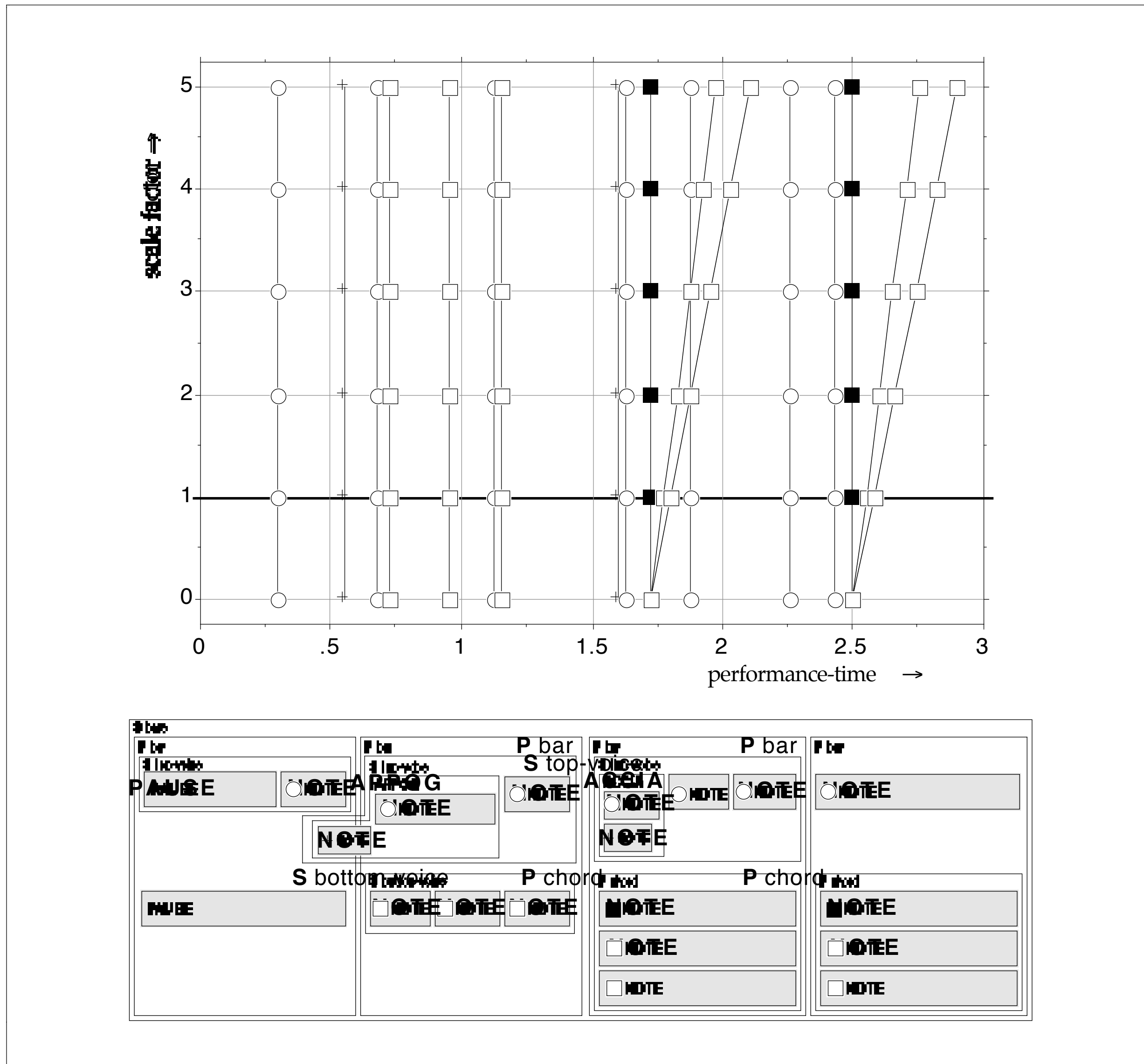

Figure 15. Scaling the expressive asynchrony of each chord in the Beethoven fragment. It shows another expressive transformation that exaggerates the chord spread, turning them almost into arpeggio's at high scale factors. At scale factor 0 the chord spread is completely removed. The timing of the rest of the fragment stays unaltered.

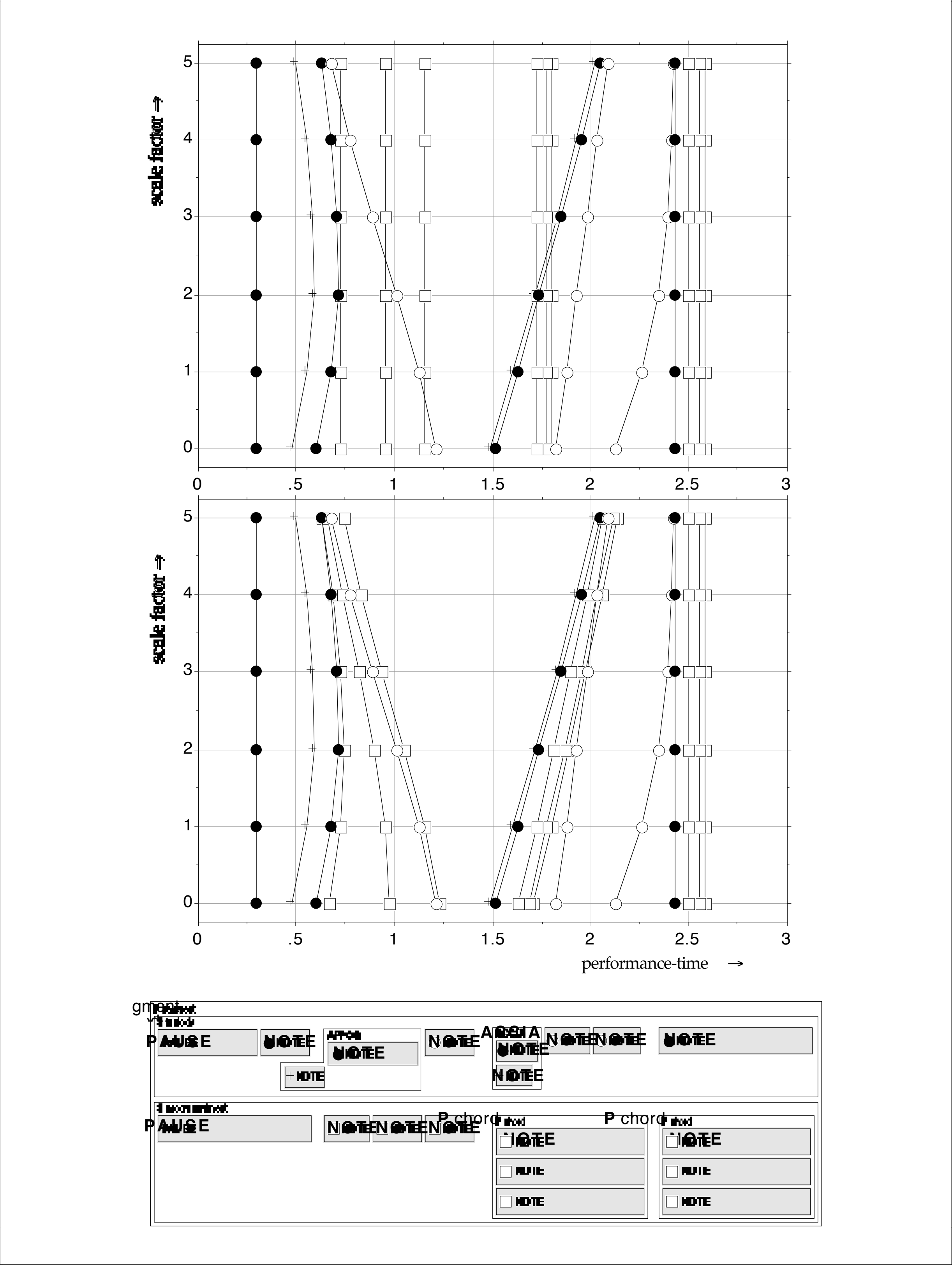

Figure 16. Scaling the expressive tempo of the melody in the Beethoven fragment, a) without and b) with "stretching" the accompaniment. It shows that the timing of each note of the melody becomes exaggerated with a higher scale factor. Here the accompaniment (lines marked with white squares) is not affected at all. Figure 16b, on the other hand, shows a musically more reasonable transformation: the accompaniment follows the movements of the transformed melody, e.g. slowing down when the tempo of the melody slows down. Here the accompaniment is kept consistent with respect to the original performance (compare with the onsets at scale factor 1). Note that note order can change between melody and accompaniment, because of the structural description in two parallel voices.

Keeping articulation consistent

In the above examples we showed the scaling of onset times and neglected what happened to the offset times. But, as we showed before, this cannot just be ignored in musically relevant transformations. We can select one of the described types of articulation to keep consistent, but we do not show this here (see figure 12 for a simple example).

Scale intervoice expression

When the expression between voices is scaled, two parameters are used. The first one selects a reference level of expression (0 designates the expression of the first, 1 designates the expression of the second, 0.5 is the mean of the two etc.). The second parameter determines in how far the voices are removed from that reference level (0 means completely on reference level, 1 means as in original performance, 2 means exaggerated with respect to the reference etc.).

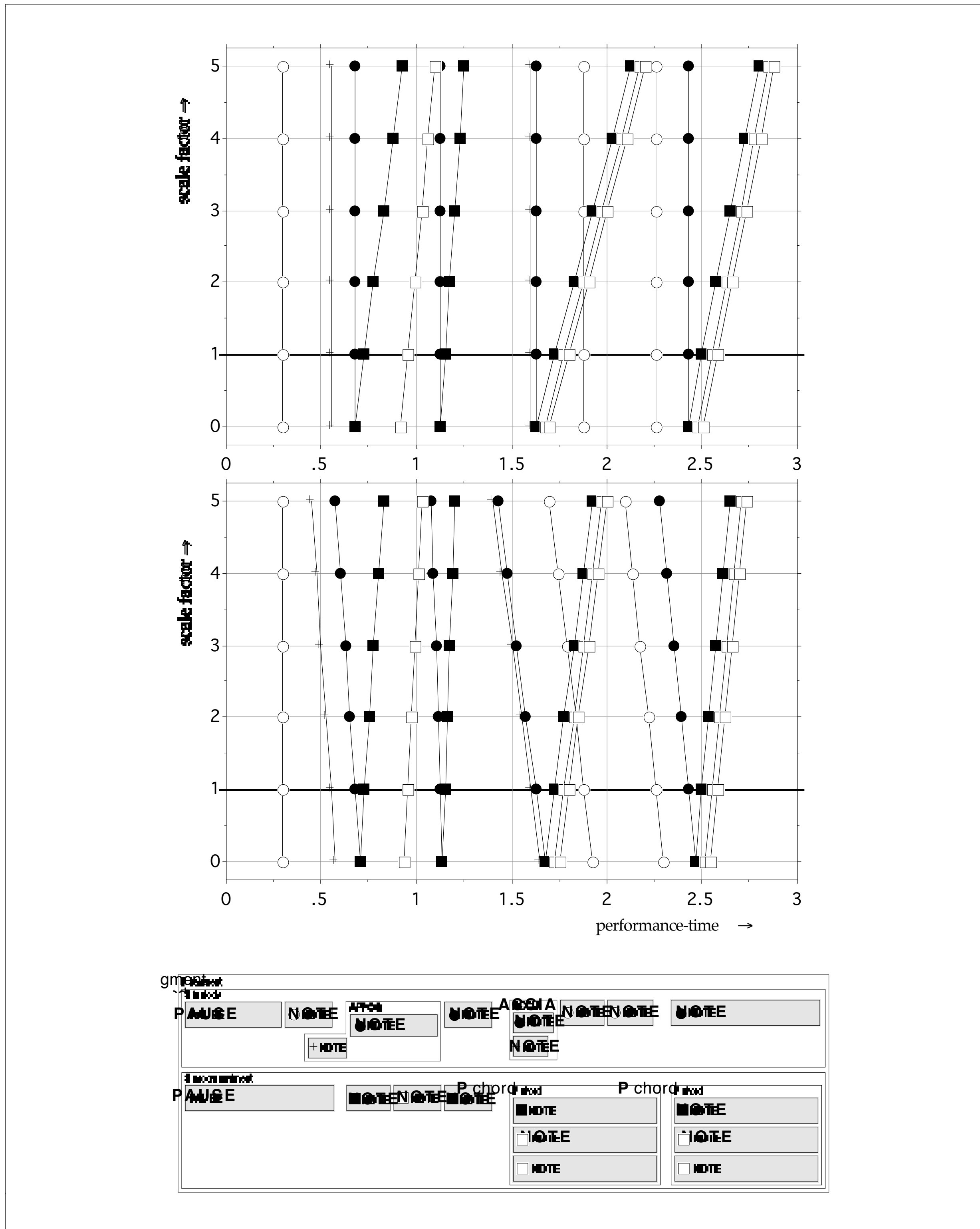

Figure 17. Scaling the intervoice timing between the melody and the accompaniment in the Beethoven fragment, a) with the melody as reference, and b) with the mean of the melody and the accompaniment as reference. In figure 17a intervoice timing (one type of intervoice scaling) is scaled with the melody voice as the reference. It shows the scaling of the asynchrony between the accompaniment and the melody, as found in the performance (see the horizontal line where the scale factor is 1). Notes that are not synchronous (i.e. don't have the same score time) interpolate their change with respect to their surrounding performance onsets that are considered parallel (have the same score time). Note that the timing of the melody does not change because it is used as reference.
In figure 17b the mean of the melody and accompaniment timing is used as reference, resulting in displacements (with respect to this invisible reference) of both voices.
In both figures, the first event in the melody voice is unaffected since there is no measurable timing in the accompaniment (only a rest).

CONCLUSION

In this paper we have presented a proposal for a calculus that enables expressive timing to be transformed on the basis of structural aspects. The program implementing the calculus, will hopefully prove to be useful for formalised theories of music cognition. The proposed representation constructs allow for easy maintenance and extension. An object-oriented programming style proved a good choice for this kind of modelling. The algorithmic parts became reasonably simple, but the program can still be considered as quite complex, especially its elaborate knowledge representation. This algorithmic simplicity combined with structural complexity mirrors, in this respect, the widespread hypothesis that the complex expressive timing profiles found in musical performances are more readily explained as the product of a small collection of simple rules linked to a relatively complex structure, than as the result of a large collection of interacting rules, with hardly any structure.

This research again confirmed that music is a very rewarding field for experimentation with knowledge representation concepts.

ACKNOWLEDGEMENTS

We are very grateful to Eric Clarke who made it possible for us to work for two years on research in expressive timing at City University in London, and the British ESRC for their financial support (grant A413254004) during this period.

REFERENCES

Bentley J. (1988) More Programming Pearls, Confessions of a Coder. Reading, MA: Addison-Wesley.

Bregman, A. S. (1990) Auditory Scene Analysis: The Perceptual Organization of Sound. Cambridge, Mass.: Bradford books, MIT Press.

Clarke, E. F. (1988) Generative principles in music performance. In J. A. Sloboda (Ed.) Generative processes in music. The psychology of performance, improvisation and composition. Oxford: Science Publications.

Desain, P. & H. Honing (1988) LOCO: A Composition Microworld in Logo. Computer Music Journal 12(3): 30-42.

Desain, P. & H. Honing (1991) Quantization of Musical Time: A Connectionist Approach. In P. M. Todd & G. J. Loy (Eds.) Music and Connectionism. Cambridge, Mass.: MIT Press.

Desain, P. & H. Honing (1992a) Tempo curves considered harmful. To appear in Contemporary Music Review.

Desain, P. & H. Honing (1992b) Time functions function best as functions of multiple times. To appear in Computer Music Journal 16(2).

Desain, P. (1990) Lisp as a second language. Perspectives of New Music 28(1).

Honing, H. (1990) POCO: An Environment for Analysing, Modifying, and Generating Expression in Music. In Proceedings of the 1990 International Computer Music Conference. San Francisco: Computer Music Association.

Honing, H. (1992) Issues in the Representation of Time and Structure in Music. In Proceedings of the 1990 Music and the Cognitive Sciences Conference, edited by I. Cross and I. Deliège. Contemporary Music Review. London: Harwood Press. (forthcoming).

Keene, S. E. (1989) Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS. Reading, MA: Addison-Wesley.

Longuet-Higgins, H. C. (1976) The Perception of Melodies. Nature 263: 646-653.

Rasch, R. A. (1979) Synchronisation in performed ensemble music. Acoustica 43, 121-131.

Serafine, M.L. (1988) Music as Cognition: The Development of Thought in Sound. New York: Columbia University Press.

Steele, G. L. (1990) Common Lisp, the Language. Second edition. Bedford, MA: Digital Press.

Vos, J. & R. A. Rasch (1981) The perceptual onset of musical tones. Perception & Psychophysics 29(4): 323-335.

APPENDIX

MICROWORLD EXPRESSION CALCULUS

```
;*************************************************************************************
;* A CALCULUS FOR MUSIC PERFORMANCE EXPRESSION                                       *
;* (c) 1991, Peter Desain & Henkjan Honing                                           *
;*                                                                                   *
;* in CLOS (Common Lisp), uses loop macro                                            *
;*************************************************************************************

;*************************************************************************************
;*************************************************************************************
; MUSICAL OBJECTS
;*************************************************************************************
;*************************************************************************************
; abstract classes of musical objects

(defclass musical-object ()
  ((name :reader name :initarg :name :initform 'no-name :type symbol)
   (score-onset :reader score-onset :type rational :initform 0)
   (left :reader left :initform nil)
   (right :reader right :initform nil))
  (:documentation "Musical Object"))

(defclass structured (musical-object)
  ((score-offset :reader score-offset :type rational))
  (:documentation "Structured Musical Object"))

(defclass multilateral (structured)
  ((components :reader components :initarg :components))
  (:documentation "Multilateral Musical Object"))

(defclass collateral (structured)
  ((main :reader main :initarg :main)
   (ornament :reader ornament :initarg :ornament))
  (:documentation "Ornamented Musical Object"))

(defclass successive (structured)
  ()
  (:documentation "Successive Musical Object"))

(defclass simultaneous (structured)
  ()
  (:documentation "Simultaneous Musical Object"))

(defclass basic (musical-object)
  ((score-offset :reader score-offset :type rational :initarg :score-dur))
  (:documentation "Basic Musical Object"))

;*************************************************************************************
; instantiatable classes of musical objects

(defclass S (multilateral successive) () (:documentation "Sequential"))
(defclass P (multilateral simultaneous) () (:documentation "Parallel"))
(defclass ACCIA (collateral simultaneous) () (:documentation "Acciaccature"))
(defclass APPOG (collateral successive) () (:documentation "Appoggiature"))

(defclass NOTE (basic)
  ((dynamic :accessor dynamic :type float :initarg :dynamic)
   (perf-onset :accessor perf-onset :type float :initarg :perf-onset :initform nil)
   (perf-offset :accessor perf-offset :type float :initarg :perf-offset :initform nil))
  (:documentation "Note"))

(defclass PAUSE (basic) () (:documentation "Rest"))
```

```lisp
;*****************************************************************************************
; creators for musical objects

(defun S (name &rest components)
  (make-instance 'S :name name :components components))

(defun P (name &rest components)
  (make-instance 'P :name name :components components))

(defun ACCIA (name ornament main)
 (make-instance 'ACCIA :name name :ornament ornament :main main))

(defun APPOG (name ornament main)
  (make-instance 'APPOG :name name :ornament ornament :main main))

(defun NOTE (&key name perf-onset perf-offset score-dur (dynamic 1))
  (make-instance 'NOTE :name name
                 :perf-onset perf-onset
                 :perf-offset perf-offset
                 :score-dur score-dur
                 :dynamic dynamic))

(defun PAUSE (&key name score-dur)
  (make-instance 'PAUSE :name name :score-dur score-dur))

;*****************************************************************************************
; extra access functions for musical objects

(defmethod components ((object basic)) nil)
(defmethod components ((object collateral))
  (list (ornament object)(main object)))

(defmethod all-notes ((object musical-object))
  (loop for component in (components object) append (all-notes component)))

(defmethod all-notes ((object note)) (list object))

(defun has-name? (&rest names)
  #'(lambda (object &rest ignore)(member (name object) names)))

(defmethod find-parts ((object musical-object) pred)
  (if (funcall pred object)
    (list object)
    (loop for component in (components object)
          append (find-parts component pred))))

;*****************************************************************************************
; initialization of score times and context of musical objects

(defmethod initialize-instance :after ((object musical-object) &rest ignore)
  (object-check object)
  (initialize-score-times object)
  (initialize-context object))

(defmethod object-check ((object musical-object)) nil)

;*****************************************************************************************
; initialization of score-onset and offset of musical objects

(defmethod initialize-score-times ((object basic)))

(defmethod initialize-score-times ((object P))
  (setf (slot-value  object 'score-offset)
        (slot-value  (first (components object)) 'score-offset)))
```

```
(defmethod initialize-score-times ((object S))
 (loop with onset = 0
        for component in (components object)
        do (shift-score component onset)
        (setf onset (slot-value  component 'score-offset))
        finally (setf (slot-value  object 'score-offset) onset)))

(defmethod initialize-score-times ((object collateral))
 (setf (slot-value  object 'score-offset)
       (slot-value  (main object) 'score-offset)))

(defmethod initialize-score-times :after ((object APPOG))
  (shift-score (ornament object)
             (- (slot-value  (ornament object) 'score-offset))))

(defmethod shift-score  ((object musical-object) shift)
  (incf (slot-value object 'score-onset) shift)
  (incf (slot-value object 'score-offset) shift)
  (loop for component in (components object) do (shift-score component shift)))

;*****************************************************************************************
; initialization of context of musical objects

(defmethod initialize-context ((object musical-object)))

(defmethod initialize-context ((object S))
  (loop for component in (components object)
        for next-component in (rest (components object))
        do (set-contexts component next-component)))

(defmethod initialize-context ((object APPOG))
  (set-context (ornament object) (main object) 'right))

(defmethod set-contexts ((left musical-object) (right musical-object))
  (set-context left right 'right)
  (set-context right left 'left))

(defmethod set-context ((object musical-object) (context musical-object) dir)
  (setf (slot-value object dir) context))

(defmethod set-context :after ((object P) (context musical-object) dir)
  (loop for component in (components object)
        do (set-context component context dir)))

(defmethod set-context :after ((object S) (context musical-object) dir)
  (if (eql dir 'left)
    (set-context (first (components object)) context dir)
    (set-context (last-element (components object)) context dir)))

(defmethod set-context :after ((object collateral) (context musical-object) dir)
  (set-context (main object) context dir))

(defmethod set-context :after ((object ACCIA) (context musical-object) dir)
  (when (eql dir 'left)
    (set-context (ornament object) context dir)))
```

```
;*********************************************************************************
;*********************************************************************************
; MAPS
;*********************************************************************************
;*********************************************************************************
; abstract classes of maps

(defclass map ()
  ((sections :accessor sections :initarg :sections))
  (:documentation "Expression Map"))

(defclass multilateral-map (map)())
(defclass collateral-map (map)())
(defclass simultaneous-map (map)())
(defclass successive-map (map)())

;*********************************************************************************
; instantiable classes of maps

(defclass P-map   (multilateral-map simultaneous-map)())
(defclass S-map   (multilateral-map successive-map)())
(defclass ACCIA-map (collateral-map simultaneous-map)())
(defclass APPOG-map (collateral-map successive-map)())

;*********************************************************************************
; creator for maps

(defun make-map (sections)
  (let ((ordered-sections (sort sections #'< :key #'score-onset)))
    (cond ((null ordered-sections) nil)
          ((and (same-section-type? ordered-sections)
                (not-overlapping? ordered-sections))
           (make-instance (section-to-map (first ordered-sections))
                          :sections ordered-sections))
          (t (error "attempt to merge incompatible sections into expression map")))))

;*********************************************************************************
; sections of maps
;*********************************************************************************
; abstract classes of sections of maps

(defclass section ()
  ((all-score-times :accessor all-score-times :initarg :all-score-times)
   (all-expressions :accessor all-expressions :initarg :all-expressions))
  (:documentation "Expression Section"))

(defclass multilateral-section (section)())
(defclass collateral-section (section)())
(defclass successive-section (section)())
(defclass simultaneous-section (section)())

;*********************************************************************************
; instantiable classes of sections of maps

(defclass S-section   (successive-section multilateral-section)())
(defclass P-section   (simultaneous-section multilateral-section)())
(defclass ACCIA-section (simultaneous-section collateral-section)())
(defclass APPOG-section (successive-section collateral-section)())

;*********************************************************************************
; compatibility relation between musical objects, expression maps and sections thereof

(defmethod object-to-section ((object musical-object))
 (third (find (class-name (class-of object)) (object-network) :key #'first)))

(defmethod section-to-map ((section section))
  (second (find (class-name (class-of section)) (object-network) :key #'third)))
```

```
(defun object-network ()
  '((S S-map S-section)
    (P P-map P-section)
    (ACCIA ACCIA-map ACCIA-section)
    (APPOG APPOG-map APPOG-section)))

;*****************************************************************************************
; creators for sections of maps

(defun make-section (section-class all-score-times all-expressions)
  (make-instance section-class
                 :all-score-times all-score-times
                 :all-expressions all-expressions))

(defmethod make-new-section ((section section) expressions)
  (make-section (class-of section)
                (snoc (score-times section) (score-offset section))
                (snoc expressions (next-expression section))))

(defmethod make-new-section-from-pairs ((section section) pairs)
  (make-section (class-of section)
                (snoc (mapcar #'first pairs) (score-offset section))
                (snoc (mapcar #'second pairs) (next-expression section))))

;*****************************************************************************************
; extra accessors for sections of maps

(defmethod score-onset ((section section))
  (first (all-score-times section)))

(defmethod score-offset ((section section))
  (last-element (all-score-times section)))

(defmethod expressions ((section section))
  (butlast (all-expressions section)))

(defmethod next-expression ((section section))
  (last-element (all-expressions section)))

(defmethod score-times ((section section))
  (butlast (all-score-times section)))

(defmethod score-onset ((section collateral-section))
  (score-main section))

(defmethod main-expression ((section collateral-section))
  (second (all-expressions section)))

(defmethod ornament-expression ((section collateral-section))
  (first (all-expressions section)))

(defmethod score-main ((section collateral-section))
  (second (all-score-times section)))

(defmethod score-ornament ((section collateral-section))
  (first (all-score-times section)))

(defun same-section-type? (sections)
  (every #'(lambda (section) (class-of section)) sections))

(defun not-overlapping? (sections)
  (loop for section in sections
        for next-section in (rest sections)
        never (> (score-offset section) (score-onset next-section))))
```

```
;*******************************************************************************
; find section (containing score time) in expression map

(defmethod lookup-section-containing ((map map) score-time)
  (loop for section in (sections map)
        when (<= (score-onset section) score-time (score-offset section))
        do (return section)))

;*******************************************************************************
; lookup expression value (via score time) in expression map

(defmethod lookup-defined-expression ((map map) score-time)
  (lookup-defined-expression (lookup-section-containing map score-time) score-time))

(defmethod lookup-defined-expression (section score-time)
  (and section
       (loop for expression in (all-expressions section)
             for map-score-time in (all-score-times section)
             when (= map-score-time score-time)
             do (return expression))))

(defmethod lookup-expression ((map successive-map) score-time)
  (lookup-expression (lookup-section-containing map score-time) score-time))

(defmethod lookup-expression (section score)
  (and section
       (loop for expression in (all-expressions section)
             for expression-next in (rest (all-expressions section))
             for score-time in (all-score-times section)
             for score-time-next in (rest (all-score-times section))
             while (> score score-time-next)
             finally (return (interpolate score-time score score-time-next
                                          expression expression-next)))))

;*******************************************************************************
; lookup score time in a monotone rising expression map

(defmethod in-section-inverse? ((section section) expression)
  (and expression (<= (first (expressions section))
                      expression
                      (or (next-expression section)
                          (last-element (expressions section))))))

(defmethod lookup-inverse ((map S-map) expression)
  (loop for section in (sections map) thereis (lookup-inverse section expression)))

(defmethod lookup-inverse ((section section) expression)
  (and (in-section-inverse? section expression)
       (loop for expression-next in (rest (expressions section))
             for score-time in (score-times section)
             for score-time-next in (rest (score-times section))
             while (> expression expression-next)
             finally (return (list score-time score-time-next)))))

;*******************************************************************************
; mapping through expression maps

(defmethod map-map (fun (map map))
  (make-map (loop for section in (sections map) collect (funcall fun section))))

;*******************************************************************************
; mapping through filtered expression maps

(defmethod with-filtered-null-expression (fun (map map))
  (unfilter-null-expression (funcall fun (filter-null-expression map))
                            (filter-null-expression-out map)))

(defmethod filter-null-expression ((map map))
  (map-map #'filter-null-expression map))

(defmethod filter-null-expression ((section section))
```

```
  (make-new-section-from-pairs section
   (loop for expression in (expressions section)
        for score-time in (score-times section)
        when expression
        collect (list score-time expression))))

(defmethod filter-null-expression-out ((map map))
  (mapcar #'filter-null-expression-out (sections map)))

(defmethod filter-null-expression-out ((section section))
  (loop for expression in (expressions section)
        for score-time in (score-times section)
        for index from 0
        unless expression
        collect (list index score-time)))

(defmethod unfilter-null-expression ((map map) rejections)
  (make-map (mapcar #'unfilter-null-expression (sections map) rejections)))

(defmethod unfilter-null-expression ((section section) removed)
  (if removed
    (make-new-section-from-pairs section
                                (loop with expressions = (expressions section)
                                      with score-times = (score-times section)
                                      for index from 0
                                      while (or score-times removed)
                                      when (and removed (= index (caar removed)))
                                      collect (list (second (pop removed)) nil)
                                      else collect (list (pop score-times)
                                                         (pop expressions))))
    section))
```

```
;*****************************************************************************************
;*****************************************************************************************
; EXPRESSION
;*****************************************************************************************
;*****************************************************************************************

(defclass expression ()())

;*****************************************************************************************
; nil and rests carry no expression, nil expressions and sections are not set

(defmethod get-expression      ((object null)(expression expression)) nil)
(defmethod get-next-expression ((object null)(expression expression)) nil)

(defmethod get-expression ((object PAUSE)(expression expression)) nil)
(defmethod set-expression ((object PAUSE)(expression expression) value) nil)

(defmethod set-expression ((object musical-object) expression value-or-section) nil)
(defmethod get-next-expression ((object musical-object)(expression expression))
  (get-expression (right object) expression))

;*****************************************************************************************
; get expression of notes

(defmethod get-notes-expression ((object musical-object) (expression expression))
  (loop for note in (all-notes object)
        collect (fetch-expression note expression)))

(defmethod set-notes-expression ((object musical-object) (expression expression) values )
  (loop for note in (all-notes object)
        for value in values
        do (set-expression note expression value)))

;*****************************************************************************************
; propagate expression (interpolated, truncating-shift and shift)

(defmethod propagate-interpolated ((object S)
                                   old-begin new-begin old-end new-end expression)
  (loop for component in (components object)
        do (propagate-interpolated component
                                   old-begin new-begin old-end new-end expression)))

(defmethod propagate-interpolated ((object P)
                                   old-begin new-begin old-end new-end expression)
  (loop for component in (components object)
        do (propagate-truncating-shift component
                                       (save-- new-begin old-begin) new-end expression)))

(defmethod propagate-interpolated ((object collateral)
                                   old-begin new-begin old-end new-end expression)
  (let* ((ref (fetch-expression (main object) expression))
         (shift (save-- (interpolate old-begin ref old-end new-begin new-end) ref)))
    (propagate-interpolated (main object) old-begin new-begin old-end new-end expression)
    (propagate-shift (ornament object) shift expression)))

(defmethod propagate-interpolated ((object NOTE)
                                   old-begin new-begin old-end new-end expression)
  (set-expression
   object expression
   (interpolate old-begin (fetch-expression object expression)
                old-end new-begin new-end)))

(defmethod propagate-interpolated ((object PAUSE)
                                   old-begin new-begin old-end new-end expression))
```

```
;*****************************************************************************************
; propagate-truncating-shift

(defmethod propagate-truncating-shift :around ((object musical-object)
                                               shift end expression)
  (when shift (call-next-method)))

(defmethod propagate-truncating-shift ((object multilateral) shift end expression)
  (loop for component in (components object)
        do (propagate-truncating-shift component shift end expression)))

(defmethod propagate-truncating-shift ((object collateral) shift end expression)
  (propagate-shift (ornament object) shift expression)
  (propagate-truncating-shift (main object) shift end expression))

(defmethod propagate-truncating-shift ((object NOTE) shift end expression)
  (set-expression object
                  expression
                  (save-min (save-+ (fetch-expression object expression) shift) end)))

(defmethod propagate-truncating-shift ((object PAUSE) shift end expression))

;*****************************************************************************************
; propagate-shift

(defmethod propagate-shift :around ((object musical-object) shift expression)
  (when shift (call-next-method)))

(defmethod propagate-shift ((object structured) shift expression)
  (loop for component in (components object)
        do (propagate-shift component shift expression)))

(defmethod propagate-shift ((object basic) shift expression)
  (set-expression object
                  expression
                  (save-+ (fetch-expression object expression) shift)))

;*****************************************************************************************
; onset timing
;*****************************************************************************************

(defclass expressive-timing (expression)        ())
(defclass onset-timing      (expressive-timing) ())
(defclass basic-asynchrony  (onset-timing)      ())
(defclass basic-tempo       (onset-timing)      ())

(defclass estimate-onset-timing (onset-timing estimate-mixin) ())

;*****************************************************************************************
; get expressive timing

(defmethod get-expression ((object NOTE) (expression onset-timing))
  (perf-onset object))

(defmethod get-expression ((object S) (expression onset-timing))
   (get-expression (first (components object)) expression))

(defmethod get-expression ((object P) (expression onset-timing))
  (loop for component in (components object)
        when (get-expression component expression)
        minimize it))

(defmethod get-expression ((object collateral) (expression onset-timing))
  (get-expression (main object) expression))
```

```
;*************************************************************************************
; set expressive timing

(defmethod set-expression ((object NOTE) (expression onset-timing) value)
  (setf (perf-onset object) value))

(defmethod set-expression ((object S) (expression onset-timing) (section S-section))
  (loop for new-expression in (expressions section)
        for next-new-expression in (snoc (rest (expressions section))
                                         (next-expression section))
        for component in (components object)
        do (propagate-interpolated component
                                   (fetch-expression component expression)
                                   new-expression
                                   (fetch-expression (right component) expression)
                                   next-new-expression
                                   expression)))

(defmethod set-expression ((object P) (expression onset-timing) (section P-section))
  (loop for new-expression in (expressions section)
        for component in (components object)
        do (propagate-truncating-shift component
                                       (save-- new-expression
                                               (fetch-expression component expression))
                                       (get-next-expression object expression)
                                       expression)))

(defmethod set-expression ((object ACCIA)
                           (expression onset-timing)
                           (section ACCIA-section))
  (propagate-shift (ornament object)
                   (save-- (ornament-expression section)
                           (fetch-expression (ornament object) expression))
                   expression))

(defmethod set-expression ((object APPOG)
                           (expression onset-timing)
                           (section APPOG-section))
  (propagate-interpolated (ornament object)
                          (fetch-expression (ornament object) expression)
                          (ornament-expression section)
                          (fetch-expression (right (ornament object)) expression)
                          (main-expression section)
                          expression))

;*************************************************************************************
; scale expressive-timing

(defmethod scale-expression ((section P-section)
                             (expression basic-asynchrony)
                             factor)
  (if (expressions section)
    (make-new-section
     section
     (scale-P-expression-points (expressions section) factor))
    section))

(defmethod scale-expression ((section S-section)
                             (expression basic-tempo)
                             factor)
  (cond ((and (expressions section)(next-expression section))
         (scale-S-section-] section factor))
        ((rest (expressions section))
         (scale-S-section-> section factor))
        (t section)))




(defmethod scale-S-section-] ((section section) factor)
    (make-new-section section  (scale-S-expression-points
```

```
                               (snoc (score-times section)(score-offset section))
                               (snoc (expressions section) (next-expression section))
                               factor)))

(defmethod scale-S-section-> ((section section) factor)
  (make-new-section section
                 (scale-S-expression-points (score-times section)
                                            (expressions section)
                                            factor)))

(defmethod scale-expression ((section ACCIA-section)
                             (expression basic-asynchrony) factor)
  (make-new-section section
                   (scale-ACCIA-points (main-expression section)
                                       (ornament-expression section)
                                       factor)))

(defmethod scale-expression ((section APPOG-section) (expression basic-tempo) factor)
  (make-new-section section
                 (scale-APPOG-points (ornament-expression section)
                                     (main-expression section)
                                     (next-expression section)
                                     (score-ornament section)
                                     (score-main section)
                                     (score-offset section)
                                     factor)))

;*****************************************************************************************

(defun scale-P-expression-points (perf-onsets factor)
  (let* ((perf-begin (apply #'min perf-onsets))
         (perf-iois (mapcar #'(lambda (onset) (- onset perf-begin)) perf-onsets))
         (raw-new-perf-iois (mapcar #'(lambda (perf)(scale-expression-lin perf factor))
                                    perf-iois))
         (shift (- (apply #'min raw-new-perf-iois)))
         (new-perf-onsets (mapcar #'(lambda (ioi) (+ ioi shift perf-begin))
                                  raw-new-perf-iois)))
    new-perf-onsets))

(defun scale-S-expression-points (score-times perf-times factor)
  (let* ((perf-iois (mapcar #'- (rest perf-times) perf-times))
         (score-iois (mapcar #'- (rest score-times) score-times))
         (perf-begin (first perf-times))
         (perf-end (last-element perf-times))
         (raw-new-perf-iois (mapcar #'(lambda (score perf)
                                        (scale-velocity score perf factor))
                                    score-iois
                                    perf-iois))
         (new-perf-iois (normalise raw-new-perf-iois (- perf-end perf-begin)))
         (new-perf-times (integrate new-perf-iois perf-begin)))
    new-perf-times))

(defun scale-ACCIA-points (main-expression ornament-expression factor)
  (let* ((expression-interval (- main-expression ornament-expression))
         (new-expression-ornament (- main-expression
                                     (scale-expression-lin expression-interval factor))))
    (list new-expression-ornament main-expression)))
```

```
(defun scale-APPOG-points  (ornament-expression main-expression next-expression
                                 score-ornament score-main score-end
                                     factor)
  (let* ((score-ornament-ioi (- score-main score-ornament ))
         (expression-ornament-ioi (- main-expression ornament-expression))
         (score-main-ioi (- score-end score-main))
         (expression-main-ioi (- next-expression main-expression))
         (ornament-tempo (/ score-ornament-ioi expression-ornament-ioi))
         (main-tempo (/ score-main-ioi expression-main-ioi))
         (relative-tempo (/ ornament-tempo main-tempo))
         (new-ornament-tempo (* main-tempo (expt relative-tempo factor)))
         (new-expression-ornament-ioi (/ score-ornament-ioi new-ornament-tempo))
         (new-expression-ornament (- main-expression new-expression-ornament-ioi)))
    (list new-expression-ornament main-expression next-expression)))

;*****************************************************************************************
; expression scale methods

(defun scale-velocity (score perf factor)
  "Exponential scaling"
  (/ score (expt (/ score perf) factor)
     ))

(defun scale-expression-lin  (perf factor)
  "Linear scaling"
  (* perf factor))

;*****************************************************************************************
; stretch expressive-timing

(defmethod stretch-expression ((section S-section)
                               (old S-map)
                               (new S-map)
                               (expression onset-timing))
  (make-new-section
   section
   (loop for perf-time in (expressions section)
         as (score-begin score-end) = (lookup-inverse old perf-time)
         collect (if (and score-begin score-end)
                   (interpolate (lookup-expression old score-begin)
                                perf-time
                                (lookup-expression old score-end)
                                (lookup-expression new score-begin)
                                (lookup-expression new score-end))
                   perf-time))))
```

```
;****************************************************************************************
; mixin to estimate expression in case of absence, by linear inter- or extrapolation
;****************************************************************************************
(defclass estimate-mixin () ())

(defmethod fetch-expression :around ((object musical-object) (expression estimate-mixin))
  (or (get-expression object expression)
      (estimate-expression object expression)))

(defmethod fetch-expression ((object null) (expression expression)) nil)

(defmethod fetch-expression ((object musical-object) (expression expression))
  (get-expression object expression))

(defmethod get-next-expression :around ((object musical-object)
                                        (expression estimate-mixin))
  (cond ((call-next-method))
        ((right object)
         (estimate-expression (right object) expression))
        (t
         (estimate-next-expression object expression))))

(defmethod fetch-onset :around ((object musical-object) (expression estimate-mixin))
  (fetch-expression object (find-expression 'estimate-onset-timing)))

(defmethod estimate-expression ((object musical-object) (expression expression))
  (estimate-context (context-with-expression object expression #'left)
                    object
                    (context-with-expression object expression #'right)
                    expression
                    t))

(defmethod estimate-next-expression ((object musical-object) (expression expression))
  (let* ((left (context-with-expression object expression #'left))
         (lefter (and left
                      (left left)
                      (context-with-expression (left left) expression #'left))))
    (when (and left lefter)
      (interpolate (score-onset lefter)
                   (score-offset object)
                   (score-onset left)
                   (get-expression lefter expression)
                   (get-expression left expression)))))

(defmethod estimate-context (left object right (expression expression) first-try)
  (cond ((and left right)
         (interpolate (score-onset left)
                      (score-onset object)
                      (score-onset right)
                      (get-expression left expression)
                      (get-expression right expression)))
        ((and left (left left) first-try)
         (estimate-context (context-with-expression (left left) expression #'left)
                           object
                           left
                           expression nil))
        ((and right (right right) first-try)
         (estimate-context right
                           object
                           (context-with-expression (right right) expression #'right)
                           expression nil))
        (t nil)))

(defmethod context-with-expression ((object musical-object)
                                    (expression expression) direction)
  (cond ((get-expression object expression)
         object)
        ((funcall direction object)
         (context-with-expression (funcall direction object) expression direction))
        (t nil)))
```

```
;***************************************************************************************
; keeping articulation invariant: mixin for expressive timing expression
;***************************************************************************************

(defclass keep-articulation-mixin () ())
(defclass keep-overlap-articulation-mixin    (keep-articulation-mixin)())
(defclass keep-duration-articulation-mixin   (keep-articulation-mixin)())
(defclass keep-proportion-articulation-mixin (keep-articulation-mixin)())

(defmethod articulation ((expression keep-overlap-articulation-mixin))
  (find-expression 'basic-overlap-articulation))

(defmethod articulation ((expression keep-duration-articulation-mixin))
  (find-expression 'basic-duration-articulation))

(defmethod articulation ((expression keep-proportion-articulation-mixin))
  (find-expression 'basic-proportion-articulation))

(defmethod set-map :around ((object musical-object)
                            map
                            (expression keep-articulation-mixin)
                            ground)
  (when map
    (let* ((parts (find-parts object ground))
           (articulation-collections
            (loop for part in parts
               collect (get-notes-expression part (articulation expression)))))
      (call-next-method)
      (loop for part in parts
            for collection in articulation-collections
            do (set-notes-expression part (articulation expression) collection))))
    object)

;***************************************************************************************
; resource for expression instances

(defvar *expression-instances*)
(setf *expression-instances* nil)
(defvar *use-expression-resource*)
(setf *use-expression-resource* t)

(defun find-expression (class)
  (or (and *use-expression-resource*
           (cdr (assoc class *expression-instances*)))
      (make-expression-instance class)))

(defun make-expression-instance (class)
  (let ((instance (make-instance class)))
    (when *use-expression-resource*
      (push (cons class instance) *expression-instances*))
    instance))

;***************************************************************************************
; averaging expression
;***************************************************************************************

(defclass averaging-expression-mixin ()())

;***************************************************************************************
; get averaging expression

(defmethod get-expression ((object multilateral) (expression averaging-expression-mixin))
  (loop for component in (components object)
        when (get-expression component expression)
        sum it into total
        finally (return (/ total (length (components object))))))

(defmethod get-expression ((object collateral) (expression averaging-expression-mixin))
  (get-expression (main object) expression))
;***************************************************************************************
; set averaging expression
```

```
(defmethod set-expression ((object multilateral)
                           (expression averaging-expression-mixin)
                           (section multilateral-section))
  (loop for component in (components object)
        for new-expression in (expressions section)
        do (propagate-shift component
                            (save-- new-expression
                                    (fetch-expression component expression))
                            expression)))

(defmethod set-expression ((object collateral)
                        (expression averaging-expression-mixin)
                        (section collateral-section))
  (propagate-shift (ornament object)
                   (save-- (ornament-expression section)
                           (fetch-expression (ornament object) expression))
                   expression))

;*****************************************************************************************
; scale averaging expression

(defmethod scale-expression ((section multilateral-section)
                             (expression averaging-expression-mixin)
                             factor)
   (let* ((mean-expression (mean (expressions section)))
          (expression-deviations (mapcar #'(lambda(expression)
                                            (- expression mean-expression))
                                         (expressions section)))
          (new-expressions
            (mapcar #'(lambda (expression-deviation)
                        (+ mean-expression
                           (scale-expression-lin expression-deviation factor)))
                                  expression-deviations)))
     (make-new-section section new-expressions)))

(defmethod scale-expression ((section collateral-section)
                             (expression averaging-expression-mixin)
                             factor)
  (let* ((expression-deviation (- (ornament-expression section)
                                  (main-expression section)))
         (new-ornament-expression
            (+ (main-expression section)
               (scale-expression-lin expression-deviation factor))))
    (make-new-section section
                      (list new-ornament-expression
                            (main-expression section)))))

;*****************************************************************************************
; stretch averaging expression

(defmethod stretch-expression ((section S-section)
                               (old S-map)
                               (new S-map)
                               (expression averaging-expression-mixin))
  (make-new-section
   section
   (loop for expression in (expressions section)
         for score-time in (score-times section)
         as old-expression = (lookup-expression old score-time)
         as new-expression = (lookup-expression new score-time)
         as stretched-expression = (if (and old-expression new-expression expression)
                                       (+ expression (- new-expression old-expression))
                                       expression)
         collect stretched-expression)))
```

```
;*************************************************************************************
; ARTICULATION
;*************************************************************************************

(defclass offset-timing                 (expressive-timing)())
(defclass articulation                  (offset-timing averaging-expression-mixin)())
(defclass basic-overlap-articulation    (articulation)())
(defclass basic-duration-articulation   (articulation)())
(defclass basic-proportion-articulation (articulation)())

;*************************************************************************************

(defmethod get-expression ((object NOTE) (expression offset-timing))
  (perf-offset object))

(defmethod fetch-onset ((object musical-object) (expression articulation))
  (get-expression object (find-expression 'onset-timing)))

;*************************************************************************************
; get articulation

(defmethod get-expression ((object NOTE) (expression basic-overlap-articulation))
   (when (right object)
    (save-- (perf-offset object)
            (fetch-onset (right object) expression))))

(defmethod get-expression ((object NOTE) (expression basic-duration-articulation))
  (- (perf-offset object)
     (fetch-onset object expression)))

(defmethod get-expression ((object NOTE) (expression basic-proportion-articulation))
  (when (and (fetch-onset object expression)
             (right object)
             (fetch-onset (right object) expression))
    (/ (- (perf-offset object)
          (fetch-onset object expression))
       (- (fetch-onset (right object) expression)
          (fetch-onset object expression)))))

;*************************************************************************************
; set articulation

(defmethod set-expression ((object NOTE) (expression basic-overlap-articulation) value)
  (when (and (right object) (fetch-onset (right object) expression))
    (setf (perf-offset object)
          (max (fetch-onset object expression)
               (+ (fetch-onset (right object) expression)
                  value)))))

(defmethod set-expression ((object NOTE) (expression basic-duration-articulation) value)
  (setf (perf-offset object)
        (+ (fetch-onset object expression)
           (max 0 value))))

(defmethod set-expression ((object NOTE)
                           (expression basic-proportion-articulation) value)
  (when (and (right object)(perf-onset (right object)))
    (setf (perf-offset object)
          (+ (fetch-onset object expression)
             (* (- (fetch-onset (right object) expression)
                   (fetch-onset object expression))
                (max 0 value))))))
```

```
;*************************************************************************************
; empty expression (to recover only score times)
;*************************************************************************************

(defclass empty-expression (expression) ())

(defmethod get-expression ((object musical-object) (expression empty-expression)) nil)

;*************************************************************************************
; mixing instantiable classes of expression
;*************************************************************************************

(defmacro class-mixer (&rest class-cocktail-pairs)
  (list* 'prog1 t
         (loop for tuples on class-cocktail-pairs by #'cdddr
               as name = (first tuples)
               as doc = (second tuples)
               as cocktail = (third tuples)
               collect `(defclass ,name ,cocktail ()
                          (:documentation ,doc)))))
```

```
(class-mixer
  tempo " "
  (basic-tempo)

  asynchrony " "
  (basic-asynchrony)

  estimate-tempo " "
  (basic-tempo estimate-mixin)

  estimate-asynchrony " "
  (basic-asynchrony estimate-mixin)

  keep-overlap-articulation-tempo " "
  (basic-tempo keep-overlap-articulation-mixin)

  keep-duration-articulation-tempo " "
  (basic-tempo keep-duration-articulation-mixin)

  keep-proportion-articulation-tempo " "
  (basic-tempo keep-proportion-articulation-mixin)

  keep-overlap-articulation-estimate-tempo " "
  (basic-tempo keep-overlap-articulation-mixin estimate-mixin)

  keep-duration-articulation-estimate-tempo " "
  (basic-tempo keep-duration-articulation-mixin estimate-mixin)

  keep-proportion-articulation-estimate-tempo " "
  (basic-tempo keep-proportion-articulation-mixin estimate-mixin)

  keep-overlap-articulation-asynchrony " "
  (basic-asynchrony keep-overlap-articulation-mixin)

  keep-duration-articulation-asynchrony " "
  (basic-asynchrony keep-duration-articulation-mixin)

  keep-proportion-articulation-asynchrony " "
  (basic-asynchrony keep-proportion-articulation-mixin)

  keep-overlap-articulation-estimate-asynchrony " "
  (basic-asynchrony keep-overlap-articulation-mixin estimate-mixin)

  keep-duration-articulation-estimate-asynchrony " "
  (basic-asynchrony keep-duration-articulation-mixin estimate-mixin)

  keep-proportion-articulation-estimate-asynchrony " "
  (basic-asynchrony keep-proportion-articulation-mixin estimate-mixin)

  overlap-articulation " "
  (basic-overlap-articulation)

  duration-articulation " "
  (basic-duration-articulation)

  proportion-articulation " "
  (basic-proportion-articulation)

  estimate-overlap-articulation " "
  (basic-overlap-articulation estimate-mixin)

  estimate-duration-articulation " "
  (basic-duration-articulation estimate-mixin)

  estimate-proportion-articulation " "
  (basic-proportion-articulation estimate-mixin))
```

```
;*****************************************************************************************
;*****************************************************************************************
; EXTRACTING AND IMPOSING EXPRESSION MAPS OF MUSICAL OBJECTS USING EXPRESSION
;*****************************************************************************************
;*****************************************************************************************
; extracting a expression map

(defmethod get-map ((object musical-object) expression ground)
  (make-map (loop for part in (find-parts object ground)
                  collect (get-section part expression))))

(defmethod get-section ((object musical-object) expression)
  (make-section (object-to-section object)
                (snoc (mapcar #'score-onset (components object))
                      (score-offset object))
                (snoc (mapcar #'(lambda (component)
                                  (fetch-expression component expression))
                              (components object))
                      (get-next-expression object expression))))

;*****************************************************************************************
; impose a expression map

(defmethod set-map ((object musical-object) map expression ground)
  (loop for part in (find-parts object ground)
        for section in (sections map)
        do (set-expression part expression section))
  object)
```

```
;*****************************************************************************************
;*****************************************************************************************
; OPERATIONS ON EXPRESSION MAPS
;*****************************************************************************************
;*****************************************************************************************
; scale expression map

(defmethod scale-map ((map map) expression factor)
  (with-filtered-null-expression #'(lambda (filtered-map)
                          (scale-filtered-map filtered-map expression factor))
                        map))

(defmethod scale-filtered-map ((map map) expression factor)
  (map-map #'(lambda (section)
               (scale-expression section
                                 expression
                                 (get-parameter factor (score-onset section))))
           map))

;*****************************************************************************************
; interpolate S-expression maps

(defmethod interpolate-maps ((map1 S-map) (map2  S-map) factor)
  (map-map  #'(lambda (section) (interpolate-section section
                                                (filter-null-expression map2)
                                                factor))

                       map1))

(defmethod interpolate-section ((section S-section)(map S-map) factor)
  (make-new-section
   section
   (loop for score-time in (score-times section)
         for expression in (expressions section)
         collect (in-between expression
                             (lookup-expression map score-time)
                             (get-parameter factor score-time)))))

(defmethod monotonise-map ((map S-map))
  (map-map #'monotonise-section map))

(defmethod monotonise-section ((section S-section))
  (make-new-section
   section
   (loop for expression in (expressions section)
         when expression
         maximize expression into state
         and collect  state
         else collect nil)))

;*****************************************************************************************
; get S-expression maps at sync points

(defmethod get-sync-map ((map1 S-map) (map2  S-map))
  (map-map  #'(lambda (section) (get-sync-section section map2)) map1))

(defmethod get-sync-section ((section S-section) (map S-map))
  (make-new-section-from-pairs  section
   (loop for score-time in (all-score-times section)
         for expression in (all-expressions section)
         as new-expression = (and expression
                                  (lookup-defined-expression map score-time))
         when new-expression collect (list score-time expression))))
```

```
;*****************************************************************************************
; stretch expression map

(defmethod stretch-map ((map successive-map)
                        (old successive-map)
                        (new successive-map)
                        expression)
  (let ((filtered-map (filter-null-expression map))
        (filtered-old (filter-null-expression old))
        (filtered-new (filter-null-expression new))
        (removed (filter-null-expression-out map)))
    (unfilter-null-expression
     (map-map
      #'(lambda (section)
          (stretch-expression section filtered-old filtered-new expression))
      filtered-map)
     removed)))

;*****************************************************************************************
;*****************************************************************************************
; TIME-CHANGING PARAMETERS
;*****************************************************************************************
;*****************************************************************************************

(defun get-parameter (factor score-time)
  (if (numberp factor)
    factor
    (funcall factor score-time)))

(defun make-ramp (x1 x2 y1 y2) ; as s-section ??
  #'(lambda (x) (interpolate x1 x x2 y1 y2)))
```

```
;*************************************************************************************
;*************************************************************************************
; TRANSFORMATIONS ON MUSICAL OBJECTS
;*************************************************************************************
;*************************************************************************************
; transfer expression transformation

(defmethod transfer ((object musical-object) expression foreground background)
  (let* ((foreground-map (get-map object expression foreground))
         (background-map (get-map object (find-expression 'empty-expression) background))
         (new-background-map (interpolate-maps background-map foreground-map 1)))
    (set-map object new-background-map expression background))
  object)

;*************************************************************************************
; scale expression transformation

(defmethod scale ((object musical-object) expression foreground background factor)
  (let* ((old-foreground-map (get-map object expression foreground))
         (new-foreground-map (when old-foreground-map
                               (scale-map old-foreground-map expression factor)))
         (old-background-map (when background
                               (get-map object expression background)))
         (new-background-map (when old-background-map
                               (stretch-map old-background-map
                                            old-foreground-map
                                            new-foreground-map
                                            expression))))
    (when new-foreground-map
      (set-map object new-foreground-map expression foreground))
    (when new-background-map
      (set-map object new-background-map expression background)))
  object)

;*************************************************************************************
; scale intervoice expression transformation

(defmethod scale-intervoice ((object musical-object) expression
                             voice1 voice2 factor ref)
  (let* ((map1 (get-map object expression voice1))
         (map2 (get-map object expression voice2)))
    (when (and map1 map2)
      (let* ((original-sync-map1 (get-sync-map map1 map2))
             (original-sync-map2 (get-sync-map map2 map1))
             (new-sync-map1 (monotonise-map (interpolate-maps
                                             original-sync-map1
                                             original-sync-map2
                                             (* ref (- 1 factor)))))
             (new-sync-map2 (monotonise-map (interpolate-maps
                                             original-sync-map2
                                             original-sync-map1
                                             (* (- 1 ref) (- 1 factor)))))
             (new-map1 (stretch-map
                        map1 original-sync-map1 new-sync-map1 expression))
             (new-map2 (stretch-map
                        map2 original-sync-map2 new-sync-map2 expression)))
        (set-map object new-map1 expression voice1)
        (set-map object new-map2 expression voice2)))
    object))
```

```lisp
;*************************************************************************************
;*************************************************************************************
; LISP UTILITIES
;*************************************************************************************
;*************************************************************************************

(defun last-element (list)
  (first (last list)))

(defun snoc (list item)
  (append list (list item)))

(defun mean (numbers)
  (/ (apply #'+ numbers) (length numbers)))

(defun save-min (&rest list)
  (let  ((new-list (remove nil list)))
    (and new-list (apply #'min new-list))))

(defun save-max (&rest list)
  (let  ((new-list (remove nil list)))
    (and new-list (apply #'max new-list))))

(defun save-- (&rest list)
  (and (notany #'null list)
       (apply #'- list)))

(defun save-+ (&rest list)
  (apply #'+ (remove nil list)))

(defun enforce-limits (minimum x maximum)
  (max minimum (min x maximum)))

(defun integrate (list start)
  (if (null list)
    (list start)
    (cons start
          (integrate (rest list) (+ (first list) start)))))

(defun normalise (list dur)
  (let ((factor  (/ dur (apply #'+ list))))
    (mapcar #'(lambda(item)(* factor item)) list)))

(defun interpolate (x1 x x2 y1 y2)
  (cond ((eql y1 y2) y1)
        ((eql x1 x2) nil)
        ((null x) nil)
        ((and x1 (= x x1)) y1)
        ((and x2 (= x x2)) y2)
        ((and x1 x2)
         (in-between y1 y2 (/ (- x x1) (- x2 x1))))
        (t nil)))

(defun in-between (y1 y2 a)
 (cond ((= a 0) y1)
       ((= a 1) y2)
       ((and y1 y2)
        (+ y1 (* a (- y2 y1))))
       (t nil)))
```

```
;*******************************************************************************
;*******************************************************************************
; EXAMPLES
;*******************************************************************************
;*******************************************************************************
#|

(defun metre-example ()
  (S 'bars
    (P 'bar
      (S 'melody
        (PAUSE :name 'pause :score-dur 1/4)
        (NOTE :name 64 :score-dur 1/8
              :perf-onset .30 :perf-offset 0.5 :dynamic .7))
      (S 'accompaniment
        (PAUSE :name 'pause :score-dur 3/8)))
    (P 'bar
      (S 'melody
        (APPOG 'appoggiatura
               (NOTE :name 64 :score-dur 1/8
                     :perf-onset .550 :perf-offset .680 :dynamic .75)
               (NOTE :name 55  :score-dur 1/4
                     :perf-onset .675 :perf-offset 1.133 :dynamic .7))
        (NOTE :name 55 :score-dur 1/8
              :perf-onset 1.125  :perf-offset 1.475 :dynamic .7))
      (S 'accompagniment
        (NOTE :name 38 :score-dur 1/8
              :perf-onset .725 :perf-offset .90 :dynamic .6)
        (NOTE :name 43 :score-dur 1/8
              :perf-onset .95 :perf-offset 1.2 :dynamic .6)
        (NOTE :name 47 :score-dur 1/8
              :perf-onset 1.150 :perf-offset 1.475 :dynamic .7)))
    (P 'bar
      (S 'melody
        (ACCIA 'acciaccatura
               (NOTE :name 59 :score-dur 1/16
                     :perf-onset 1.600 :perf-offset 1.7 :dynamic .65)
               (NOTE :name 57 :score-dur 1/8
                     :perf-onset 1.625 :perf-offset 1.880 :dynamic .7))
        (NOTE :name 55 :score-dur 1/8
              :perf-onset 1.880 :perf-offset 2.256 :dynamic .6)
        (NOTE :name 57 :score-dur 1/8
              :perf-onset 2.256 :perf-offset 2.647 :dynamic .65))
      (S 'accompagniment
        (P 'chord
          (NOTE :name 38 :score-dur 3/8
                :perf-onset 1.725 :perf-offset 2.500 :dynamic .7)
          (NOTE :name 42 :score-dur 3/8
                :perf-onset 1.775 :perf-offset 2.500 :dynamic .65)
          (NOTE :name 48 :score-dur 3/8
                :perf-onset 1.800 :perf-offset 2.500 :dynamic .7))))
    (P 'bar
      (S 'melody
        (NOTE :name 55 :score-dur 3/8
              :perf-onset 2.425 :perf-offset 4 :dynamic .7))
      (S 'accompagniment
        (P 'chord
          (NOTE :name 43 :score-dur 3/8
                :perf-onset 2.500 :perf-offset 4 :dynamic .6)
          (NOTE :name 47 :score-dur 3/8
                :perf-onset 2.550 :perf-offset 4 :dynamic .7)
          (NOTE :name 50 :score-dur 3/8
                :perf-onset 2.580 :perf-offset 4.5 :dynamic .65))))))
```

```
(defun background-example ()
 (P 'fragment
     (S 'melody
        (PAUSE :name 'pause :score-dur 1/4)
        (NOTE :name 64 :score-dur 1/8
              :perf-onset 0.3 :perf-offset 0.5 :dynamic .7)
        (APPOG 'appoggiatura
               (NOTE :name 64 :score-dur 1/8
                     :perf-onset .550 :perf-offset .680 :dynamic .75)
               (NOTE :name 55  :score-dur 1/4
                     :perf-onset .675 :perf-offset 1.133 :dynamic .7))
        (NOTE :name 55 :score-dur 1/8
              :perf-onset 1.125  :perf-offset 1.475 :dynamic .7)
        (ACCIA 'acciaccatura
               (NOTE :name 59 :score-dur 1/16
                     :perf-onset 1.600 :perf-offset 1.700 :dynamic .65)
               (NOTE :name 57 :score-dur 1/8
                     :perf-onset 1.625 :perf-offset 1.880 :dynamic .7))
        (NOTE :name 55 :score-dur 1/8
              :perf-onset 1.880 :perf-offset 2.256 :dynamic .6)
        (NOTE :name 57 :score-dur 1/8
              :perf-onset 2.256 :perf-offset 2.647 :dynamic .65)
        (NOTE :name 55 :score-dur 3/8
              :perf-onset 2.425 :perf-offset 4 :dynamic .7))
     (S 'accompagniment
        (PAUSE :name 'pause :score-dur 3/8)
        (NOTE :name 38 :score-dur 1/8
              :perf-onset .725 :perf-offset .90 :dynamic .6)
        (NOTE :name 43 :score-dur 1/8
              :perf-onset .950 :perf-offset 1.2 :dynamic .6)
        (NOTE :name 47 :score-dur 1/8
              :perf-onset 1.150 :perf-offset 1.475 :dynamic .7)
        (P 'chord
           (NOTE :name 38 :score-dur 3/8
                 :perf-onset 1.725 :perf-offset 2.500 :dynamic .7)
           (NOTE :name 42 :score-dur 3/8
                 :perf-onset 1.775 :perf-offset 2.500 :dynamic .65)
           (NOTE :name 48 :score-dur 3/8
                 :perf-onset 1.800 :perf-offset 2.500 :dynamic .7))
        (P 'chord
           (NOTE :name 43 :score-dur 3/8
                 :perf-onset 2.500 :perf-offset 4 :dynamic .6)
           (NOTE :name 47 :score-dur 3/8
                 :perf-onset 2.550 :perf-offset 4 :dynamic .7)
           (NOTE :name 50 :score-dur 3/8
                 :perf-onset 2.580 :perf-offset 4.5 :dynamic .65)))))

;data at factor 2 in figure 13
(scale (metre-example)
       (find-expression 'tempo) (has-name? 'bars) nil
       2)

;data at factor 2 in figure 14
(scale (metre-example)
       (find-expression 'asynchrony)  (has-name? 'bar) nil
       2)

;data at factor 2 in figure 16a
(scale (background-example)
       (find-expression 'tempo) (has-name? 'melody) nil
       2)

;data at factor 2 in figure 16b
(scale (background-example)
       (find-expression 'tempo) (has-name? 'melody) (has-name? 'accompaniment)
       2)

|#
```