ISSN 1744-1986

Technical Report N^{o} 2006/12

Tutorial : An Introduction to AspectMusic

Pat Hill
Simon Holland
Robin Laney

12th October 2006

Department of Computing
Faculty of Mathematics and Computing
The Open University
Walton Hall,
Milton Keynes
MK7 6AA
United Kingdom

http://computing.open.ac.uk

Tutorial : An Introduction to AspectMusic

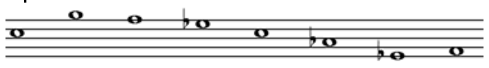
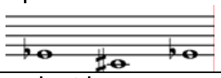
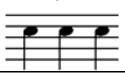
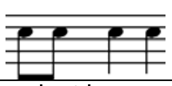
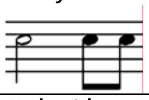
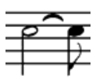
This tutorial is intended to be read in conjunction with the paper "An Introduction to Aspect-Oriented Music Representation". In this tutorial we show, by means of concrete examples, some of the ways in which AspectMusic may be used to represent and compose CMUs through HyperMusic, and how CMUs may be arranged, and crosscutting musical concerns expressed and applied using MusicSpace.

1. HyperMusic

In this section, we demonstrate some of the ways in which HyperMusic allows musical material to be represented, organised, and composed together to form new materials. Note that, as we will demonstrate, HyperMusic permits the user to work across multiple, possibly incomplete dimensions of musical concern. Consequently, the approach we describe in this tutorial is just one of many possibilities.

Throughout this tutorial, we will use examples drawing from an informal analysis of Boellmann's "Priere a Notre Dame". We will assume that the following fragments have been organised into a hyperspace as follows. Again, of course, the organisation used here is just one possibility.

Hyperspace

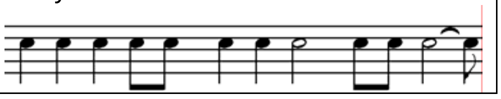
Dimension	Concern	Name	CMU
`Section1`	`Themes`	`Melody1`	`#pitch` 
`Section1`	`Themes`	`Melody2`	`#pitch` 
`Section1`	`Rhythms`	`R1`	`#rhythm` 
`Section1`	`Rhythms`	`R2`	`#rhythm` 
`Section1`	`Rhythms`	`R3`	`#rhythm` 
`Section1`	`Rhythms`	`R4`	`#rhythm` 
`Transformations`	`Pitch`	`Transpose`	A transform that transposes pitch by some value given in the parameter named `Delta`

1.1 A Melodic Line and Variations

Imagine that we are composing a melody line. We have decided that the rhythm of our melody will consist of the elements `R1..R4,` in sequence. We have also decided on the opening of the melody (`Melody1`), but we want to have two variations of this melody, each with different endings.

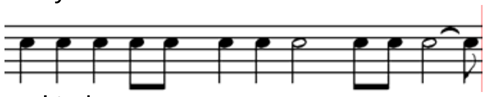
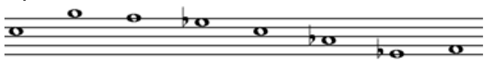
First, we can build a CMU that represents the rhythm of the melody, we will call this `Section1;Motives;Rhythm`.

```
Hypermodule:
      Section1;Motives;Rhythm
Hyperslices:
      Section1;Rhythms;
Relationships:
      MergeByName
CompositionExpression:
      .*;.*;.*
```

Dimension	Concern	Name	CMU
Section1	Motives	Rhythm	#rhythm 

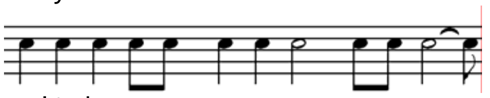
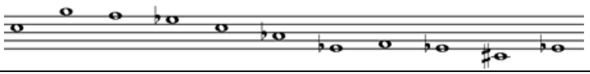
We can now create a CMU `Section1;Variations;MelodyBase` that adds the opening part of the melody to our rhythm. Note that there are less pitch elements than rhythm elements. Note also, that the pitch element does not align with the rhythmic fragments `R1..R4`, rather it ends at the third crotchet beat of `R3`.

```
Hypermodule:
      Section1;Variations;MelodyBase
Hyperslices:
      .*;.*;
Relationships:
      MergeByName
CompositionExpression:
      Section1;Motives;Rhythm + Section1;Themes;Melody1;
```

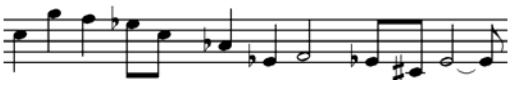
Dimension	Concern	Name	CMU
Section1	Variations	MelodyBase	#rhythm  #pitch 

The first variation can be constructed by adding `Section1;Themes;Melody2` as an ending to our `MelodyBase` CMU.

```
Hypermodule:
      Section1;Variations;Melody1
Hyperslices:
      .*;.*;
Relationships:
      MergeByName
CompositionExpression:
      Section1;Variations;MelodyBase + Section1;Themes;Melody2;
```

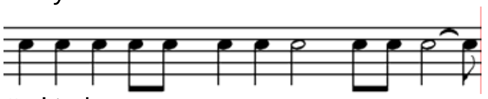
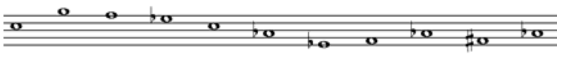
Dimension	Concern	Name	CMU
Section1	Variations	Melody1	#rhythm  #pitch 

The interpretation of `Melody1` is as follows.

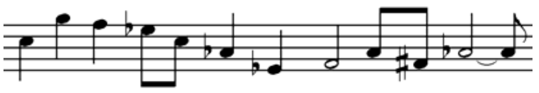


Another variation may be formed by using the `Transpose` transform to transpose `Melody2` up by five semitones prior to adding it to `MelodyBase`.

```
Hypermodule:
      Section1;Variations;Melody2
Hyperslices:
      .*;.*;
Relationships:
      MergeByName
CompositionExpression:
      Section1;Variations;MelodyBase + @(Section1;Themes;Melody2 +
      Transforms;Pitch;Transpose[Delta=5])
```

Dimension	Concern	Name	CMU
Section1	Variations	Melody2	#rhythm  #pitch 

`Melody2` is interpreted as follows



1.2 Expressing Structure with CMUs.

We may now decide that the variations `Melody1` and `Melody2`, in sequence, are enough to form the opening of section 1 of our piece, in the melodic dimension at least. We can specify a CMU that represents this arrangement.

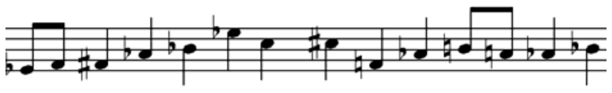
```
Hypermodule:
      Section1;Opening;Melody
Hyperslices:
      Section1;Variations
Relationships:
      MergeByName
CompositionExpression:
      .*;.*;Melody[12];
```

Alternatively, we could decide that the opening melody arrangement may consist of a number of variations; we've defined two of them, but we may come back later to define others. The following CMU definition allows us to do this. Due to the hyperslice specification, the hypermodule specification will find only those melody variations (CMUs) whose names begin with "`Melody`" and that are defined in the `Variations` concern of the `Section1` dimension. Thus within a single hyperspace, we can construct other CMUs called "`Melody`" in other dimensions and concerns, without affecting `Section1`. This composition expression will assemble the variations sequentially, in ascending name order.

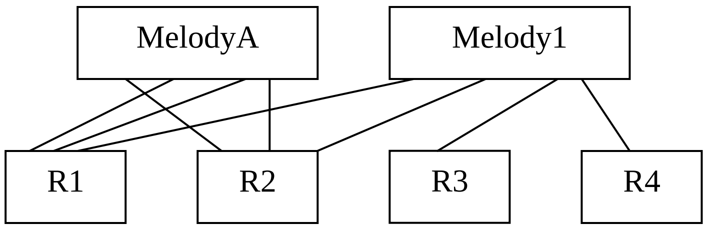
```
Hypermodule:
      Section1;Opening;Melody
Hyperslices:
      Section1;Variations
Relationships:
      MergeByName
CompositionExpression:
      .*;.*;Melody.*;
```

1.3 Representing Polyarchic Relationships

Consider the following fragment (`MelodyA`) from "Priere".



The rhythmic elements `R1` and `R2` are shared between the rhythmic hierarchies of `Melody1` and `MelodyA`. Thus there are polyarchic relationships between rhythmic components of `MelodyA` and those of the melodic fragments (such as `Melody1`) defined above. This is shown below with the ordering of elements shown left-to-right.



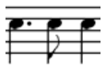
HyperMusic does not explicitly support polyarchic relationships, however, since all CMUs are available for further composition at any level of abstraction, such relationships may be expressed implicitly. We can, for example, consider the rhythm of `MelodyA` to be composed as follows.

```
Hypermodule:
      Section1;MelodyA;Rhythm
Hyperslices:
      Section1;Rhythms
Relationships:
      MergeByName
CompositionExpression:
    .*;.*;R2 + .*;.*;R1 + .*;.*;R1 + .*;.*;R2
```

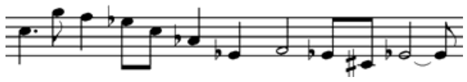
1.4 Sameness Relationships in HyperMusic

As the previous example shows, HyperMusic enables relationships to be established between arbitrary fragments and higher-level composites. Superficially, these relationships mean that, for example, the `R1` in `Melody1` and those in `MelodyA` are the same. In HyperMusic, the sameness relationship, is established by means of the appearance of the same CMU in multiple composition expressions, or multiple appearances of a CMU in the same composition expression. Sameness means that a CMU a) *has the same content* in all of the CMUs in which it is a component and that b) if a change is made to a component CMU, then that change will be reflected in *all* the CMUs in which it features.

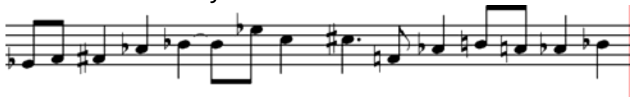
So, for instance, if `R1` was changed to



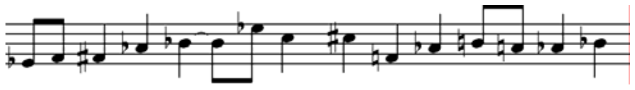
Then `Melody1` would change correspondingly



as would `MelodyA`



But say we wanted the second instance of `R1` to remain unchanged in `MelodyA`.



This means that the original expression for the rhythmic element of `MelodyA` was incorrect, because the two three-crotchet rhythmic motifs, while having the same content, are actually unrelated. In other words, while the sameness condition a) is true, condition b) does not hold. In HyperMusic, only the hyperspace coordinate is constrained to be unique within a hyperspace. Consequently, units with identical content are permitted. Thus we could define an additional rhythmic element (`R5` say), having initially the same three crotchet content as `R1`, and compose the rhythmic component of `MelodyA` as

```
R2 + R1 + R5 + R2
```

The dynamic nature of HyperMusic enables us to make these alterations, namely the insertion of `R5` into the hyperspace and the re-definition of `MelodyA`, at any time, though of course, `MelodyA` and any dependent CMUs would need to be recompiled.

1.5 Composition History

HyperMusic supports the production of “Composition History”. In our current implementation, a history record is added to each affected element of each `MusicItemCollection` of each MU within a CMU when:

A CMU is woven through hypermodule specifications
A CMU is populated by importing content from a (MIDI) file
A CMU is added to a hyperspace
A CMU is transformed

Primarily, composition history is intended to be used with MusicSpace, which we discuss in the next section. However, composition history also provides a useful diagnostic tool. A composer may, for example, examine the content of a CMU and use composition history to identify the derivation of particular musical elements, thus providing the ability to “debug” a musical piece as it evolves.

The following table shows a representation of a composition history associated with a single `MusicUnitItem`.

Index	History Event	Parameters
1	`#Binding`	`#filename=’MyFile.MID’, #track=2`
2	`#HyperspaceLocation`	`#MU=#pitch, #CMUIndex=3,` `#Location=(#MyTune, #Pitch, #Demo)`
3	`#Transform`	`#args=(#Delta=-12, #UnitType=#pitch)` `#class=#TransposeTransform`
4	`#HyperspaceLocation`	`#MU=#pitch #CMUIndex=3` `#Location=(#newUnits, #test, #newUnit)`

This history tells us that the `MusicUnitItem` with which it is associated originated in track 2 of a file called “`MyFile.MID`”. The item was composed into a CMU called “`Demo`”, which was in the “`Pitch`” concern of the “`MyTune`” dimension in the hyperspace, and the item was located at index 3 of the `#pitch` MU. The item was subsequently transformed by a `TransposeTransform`, which was given the parameters `#Delta=-12`, `UnitType=#pitch`, and the newly formed CMU was added to the hyperspace as “`newUnit`” in the “`test`” concern of the “`newUnits`” dimension. The item remained at index 3 in the `#pitch` MU of the new CMU.

The current value of the item may be determined from the item itself, while the HyperspaceLocation events within the composition history provide sufficient information to determine any previous values that were held by the item at any stage in its composition, by querying the hyperspace.

1.6 Summary

In these examples, we have demonstrated a number of features of HyperMusic. In particular:

1) HyperMusic enables musical information to be organised in arbitrary ways that are meaningful to the user.

2) HyperMusic permits the user to work flexibly, with incomplete musical ideas and at different levels of abstraction.
3) HyperMusic allows the user to work with tangled hierarchies and polyarchies across different dimensions.
4) HyperMusic abstracts composition and transformational processes from musical content.
5) The hyperslice feature of HyperMusic enables the user to partition the hyperspace such that related elements are isolated and protected from changes occurring elsewhere in the hyperspace.
6) The use of regular expressions in composition expressions facilitates the expression of open-ended compositions.
7) Hypermodules facilitate the rapid construction of musical materials, facilitating experimentation and the exploration of "what-if" scenarios.
8) Composition History provides a detailed history of the derivation of each element within a CMU.

2. MusicSpace

MusicSpace enables CMUs to be arranged sequentially in MusicSpaceParts, and for MusicSpaceParts to be arranged in parallel with each other in a manner similar to a typical MIDI sequencer. The main purpose of MusicSpace is to provide an environment in which musical material obtained from CMUs may be modified based upon context, such as temporal or metrical location, or other CMUs or specific events which occur simultaneously. Such modification processes are expressed as MusicSpace Aspects, which encapsulate both the modification itself, and expressions that identify all the conditions that must be met in order for the modification to be applied. Like their counterparts in systems such as AspectJ, MusicSpace Aspects enable crosscutting concerns to be modularised.

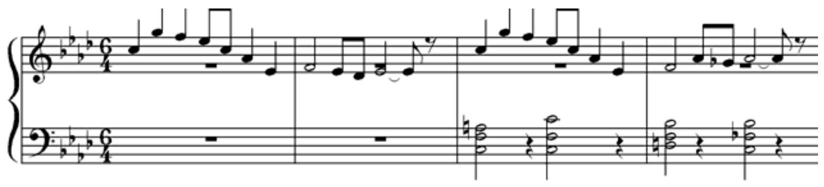
To recap, in the MusicSpace system, a populated MusicSpace is compiled against an `AspectManager` with which all required MusicSpace Aspects are registered. The compilation produces a new MusicSpace instance. The compilation process involves issuing a clock 'tick' which advances through the MusicSpace. At each tick a `MusicSpaceJoinpointContext` is generated, containing all the events within each MusicSpacePart that start at that tick.

Each MusicSpace aspect defines a pair of pointcuts and advice, termed *before* and *after*. Before-pointcuts are evaluated immediately the `MusicSpaceJoinpointContext` becomes available. If the pointcut evaluates to true, then the before-advice is called, passing the `MusicSpaceJoinpointContext` to the advice. The advice may inspect and modify the both the joinpoint context, and the MusicSpace itself. Note that if future events are added to the MusicSpace then they will form part of a future `MusicSpaceJoinpointContext`.

Once all before-advices have run, the joinpoint context is copied back to the new MusicSpace instance, at the same location as the joinpoint context itself. Once the joinpoint context has been copied, after-pointcuts on all registered aspects are evaluated, and if true, the associated after-advice is executed. After-advice can inspect the modified joinpoint context, and modify the MusicSpace, however while after-advice may modify the joinpoint context, no such modifications will be reflected in the new MusicSpace instance.

2.1 Constructing and Populating a MusicSpace

In this section we will illustrate four types of pointcuts that can be expressed using MusicSpace. We use a common example throughout the illustrations. For this example, we set up a MusicSpace containing two parts. The first part (`part1`) contains the two melody variations that were constructed in the HyperMusic tutorial above. The CMUs that contains these variations are respectively called `var1` and `var2`. The second part (`part2`) contains a harmonic figure (`harm1`), which we want to accompany only the second variation. The content of the MusicSpace is shown, in common practice notation, below.



We can define this arrangement of parts, statically, as follows. Here we have defined the `harm1` figure to start at bar 3, coinciding with the metrical position of `var2`.

```
mspace := MusicSpace new.

part1 := mspace createPart: #part1.
part2 := mspace createPart: #part2.
part1 startingAt: (CuePoint new bar: 1 beat: 1 tick: 0) put: (var1
value).
part1 startingAt: (CuePoint new bar: 3 beat: 1 tick: 0) put: (var2
value).
part2 startingAt: (CuePoint new bar: 3 beat: 1 tick: 0) put: (harm1
value).
```

The following illustrations consider various ways in which a “Staccato” aspect may be applied. For the purposes of this discussion, assume that the before-advice of the staccato aspect halves the duration of any events in part1 that are present in the joinpoint context.

2.2 Temporal Pointcuts

Consider the case where we want to apply the staccato aspect to `var2`. One way to achieve this is to use a temporal pointcut that causes the advice to be applied based upon metrical location. Each part within a joinpoint exposes a cuepoint object from which the current bar, beat and beat subdivision (tick) may be obtained. Of course, if each part is configured with the same time signature, then the cuepoints for each part will be the same. In the following example, we define the before pointcut of the `StaccatoAspect` such that it returns true, and thereby causes the associated before advice to run, when the bar number is greater than 2 and less than 4. In other words, the aspect is applied for the duration of `var2`.

```
staccatoAspect := StaccatoAspect new.
```

```
staccatoAspect
      beforePointcut:
            [:aspect :ctx |
                  part1Event := ctx at: #part1.
                  part1Event = nil
                        ifTrue: [false]
                        ifFalse: [part1Event cuepoint bar > 2 and:
                          [part1Event cuepoint < 4]].
```

2.3 History-based Pointcuts

The pointcut shown above will apply the Staccato aspect's before advice to the content of part1 when the joinpoint, according to `part1`, is in bar 3 or greater. Since `var2` is located at bar 3, its content will be made staccato. However, aside from the fact that, in this example, the second variation (`var2`) happens to be at this location, there is no relationship between `var2` and the Staccato aspect. Thus the aspect above will apply its advice to *whatever* content is placed between bars 2 and 4. If we want the staccato to be applied to `var2` independently of its location, then we must associate the staccato aspect with `var2`.

One way to associate an aspect with particular events is to use event history. This approach enables us, for example, to express joinpoint conditions that are satisfied if an event is currently part of, or has been derived from, a particular CMU, or has been transformed in some particular way,

The following pointcut specification queries the history of each event in part1 in order to determine whether or not the event belongs to a CMU whose hyperspace location was `Section1;Variations;Variation2`. In this case, we are interested in HyperspaceLocation events, and in particular, the most recent `HyperspaceLocation` event. The `mostRecentItemForEvent:` selector enables us to retrieve the most recent history item for some given history event. Consequently, this pointcut would intentionally not pick out events that have been *derived* from `Variation2` and placed into a CMU elsewhere in the hyperspace.

```
staccatoAspect beforePointcut:
  [:aspect :ctx |
    part1Event := ctx at: #part1.
    part1Event size > 0
    ifTrue:
      [cmuEvent := part1Event events at: 1.
      pitchItemSet := cmuEvent at: #pitch.
      pitchItem := pitchItemSet at: 1.
      history := pitchItem history.
      locationDetails := history
      mostRecentItemForEvent: #HyperspaceLocation.
      hyperspaceLocation := locationDetails at: #Location.
            ((hyperspaceLocation name = #Variation2 )
            and: [ hyperspaceLocation concern = #Variations
            and: [ hyperspaceLocation dimension = #Section1]])
      ifTrue:
        [true]
      ifFalse:
        [false]]
    ifFalse: [false]].
```

2.4 Context-based Pointcuts

The previous examples have respectively shown how pointcuts may be written to associate advice with metrical location and events that have particular history components. In this example, we consider associating aspects with events based upon the context of the event. In this example, we want to apply the staccato aspect to events in part1, but only if those events *coincide* with events in `part2`.

The following simple pointcut will achieve this effect. The pointcut simply states that if there are any events in part2 at this joinpoint, then apply the advice.

```
staccatoAspect beforePointcut:
      [:aspect :ctx |
             (ctx at: #part2) size > 0].
```

Recall that the advice is only applied to events in the joinpoint context from `part1`. The overall effect therefore is modify the duration of any event in `part1` within the joinpoint context when the joinpoint context also contains an event in `part2`.

2.5 Content-Based Pointcuts

The previous examples have demonstrated pointcuts that relate to metrical location, event derivation through composition history, and the context in which an event occurs within the MusicSpace. The final pointcut type that we will illustrate is a content-based pointcut. This type of pointcut is used to pick out events where the information contained within one or more events within the joinpoint context is of interest.

Since it cannot be known, a priori, what types of information are of interest or what relationships must exist between these items, it is difficult to provide any explicit support within the MusicSpace classes to facilitate extracting this information. Rather, the user must adopt a more general approach involving navigating the objects within the joinpoint context, identifying and extracting those items of interest and performing appropriate comparisons to determine whether or not the pointcut is satisfied. Consequently, expressing these kinds of pointcut through the Smalltalk interface is arbitrarily complex. The logic-based representation of a joinpoint context provides a more compact and accessible form. Moreover, the task of determining whether a joinpoint context satisfies some arbitrary condition can be delegated to SOUL. In the following examples, we will illustrate the use the logic-based representation.

The following is an example of a joinpoint context represented as SOUL facts. The representation is split into three fact types. Event facts detail the content of all the events in all of the parts represented in the joinpoint context. History facts detail the symmetric composition history of all events in all parts represented in the joinpoint context. Finally, cuepoint facts describe the current metrical location for each part that is represented in the joinpoint context.

```
event(part1,rhythm,1,value(<onset(0),duration([480]),
      rest(false)>))
event(part1,pitch,1,value(<midiPitch(72)>))
event(part2,rhythm,1,value(<onset(0),duration[2880], rest(false)>))
event(part2,pitch,1,value(<midiPitch(51)>))
event(part2,pitch,2,value(<midiPitch(60)>))
event(part2,pitch,3,value(<midiPitch(56)>))
history(part1,rhythm,1,1,Binding(<parm(filename,{priereR1.mid}),
      parm(track,1)>))
history(part1,rhythm,1,2,HyperspaceLocation(<parm(MUType,{rhythm}),
      parm(dimension,{Section1}),parm(concern,{Rhythms}),
      parm(unit,{R1}),parm(CMUIndex,1)>))
history(part1,rhythm,1,3,HyperspaceLocation(<parm(MUType,{rhythm}),
      parm(dimension,{Section1}),parm(concern,{Motives}),
      parm(unit,{Rhythm}),parm(CMUIndex,1)>))
history(part1,rhythm,1,4,HyperspaceLocation(<parm(MUType,{rhythm}),
      parm(dimension,{Section1}),parm(concern,{Variations}),
      parm(unit,{MelodyBase}),parm(CMUIndex,1)>))
history(part1,rhythm,1,5,HyperspaceLocation(<parm(MUType,{rhythm}),
      parm(dimension,{Section1}),parm(concern,{Variations}),
      parm(unit,{Variation2}),parm(CMUIndex,1)>))
history(part1,pitch,1,1,Binding(<parm(filename,{priereM1.mid}),
      parm(track,1)>))
history(part1,pitch,1,2,HyperspaceLocation(<parm(MUType,{pitch}),
      parm(dimension,{Section1}),parm(concern,{Themes}),
      parm(unit,{Melody1}),parm(CMUIndex,1)>))
history(part1,pitch,1,3,HyperspaceLocation(<parm(MUType,{pitch}),
      parm(dimension,{Section1}),parm(concern,{Variations}),
      parm(unit,{MelodyBase}),parm(CMUIndex,1)>))
history(part1,pitch,1,4,HyperspaceLocation(<parm(MUType,{pitch}),
      parm(dimension,{Section1}),parm(concern,{Variations}),
      parm(unit,{Variation2}),parm(CMUIndex,1)>))
history(part2,rhythm,1,1,Binding(<parm(filename,{priereHR1.mid}),
      parm(track,1)>))
history(part2,rhythm,1,2,HyperspaceLocation(<parm(MUType,{rhythm}),
      parm(dimension,{Section1}),parm(concern,{Harmony}),
      parm(unit,{R1}),parm(CMUIndex,1)>))
history(part2,rhythm,1,3,HyperspaceLocation(<parm(MUType,{rhythm}),
      parm(dimension,{Section1}),parm(concern,{Harmony}),
      parm(unit,{HR1}),parm(CMUIndex,1)>))
history(part2,pitch,1,1,Binding(<parm(filename,{priereHC2.mid}),
      parm(track,2)>))
history(part2,pitch,1,2,HyperspaceLocation(<parm(MUType,{pitch}),
      parm(dimension,{Section1}),parm(concern,{Harmony}),
      parm(unit,{Progression1}),parm(CMUIndex,1)>))
history(part2,pitch,1,3,HyperspaceLocation(<parm(MUType,{pitch}),
      parm(dimension,{Section1}),parm(concern,{Harmony}),
      parm(unit,{HR1}),parm(CMUIndex,1)>))
history(part2,pitch,2,1,Binding(<parm(filename,{priereHC2.mid}),
      parm(track,2)>))
history(part2,pitch,2,2,HyperspaceLocation(<parm(MUType,{pitch}),
      parm(dimension,{Section1}),parm(concern,{Harmony}),
      parm(unit,{Progression1}),parm(CMUIndex,1)>))
history(part2,pitch,2,3,HyperspaceLocation(<parm(MUType,{pitch}),
      parm(dimension,{Section1}),parm(concern,{Harmony}),
      parm(unit,{HR1}),parm(CMUIndex,1)>))
history(part2,pitch,3,1,Binding(<parm(filename,{priereHC2.mid}),
      parm(track,2)>))
history(part2,pitch,3,2,HyperspaceLocation(<parm(MUType,{pitch}),
      parm(dimension,{Section1}),parm(concern,{Harmony}),
      parm(unit,{Progression1}),parm(CMUIndex,1)>))
history(part2,pitch,3,3,HyperspaceLocation(<parm(MUType,{pitch}),
      parm(dimension,{Section1}),parm(concern,{Harmony}),
      parm(unit,{HR1}),parm(CMUIndex,1)>))
cuepoint(part1,location(3,1,0))
cuepoint(part2,location(3,1,0))
```

The representation may be queried using SOUL queries. For some applications, only the existence of a solution is important. In others, the details of the solution(s) may be required.

The following queries are illustrative:

The following query returns a result if `part1` contains a rhythmic value with a duration less than 480 ticks.

```
event(part1,rhythm,?,value(?P)), member(duration(?X),?P),
smaller( ?X, [480])
```

The following query returns a result if `part2` contains an interval of a perfect fourth (5 semitones)

```
event(part2,pitch,?E1,value(?P)), member(midiPitch(?X),?P),
event(part2,pitch,?E2,value(?Q)), member(midiPitch(?Y),?Q),
sub(?X,?Y,?Z),equals(?Z,5)
```

Using SOUL, it is possible not only to determine whether the pointcut is satisfied, but also, what values each variable in the query must take in order to satisfy the query. This, given the joinpoint context and the query shown above, we can discover that not only is there an interval of a perfect fourth, but that this interval occurs between elements with MIDI pitch values of 56 (`?X`) and 51 (`?Y`), and that these elements are respectively at indexes 3 (`?E1`) and 1 (`?E2`).

2.6 Linking with HyperMusic

The joinpoint context produced by MusicSpace contains only the current values of the events it contains. While it is possible to discover the event history of a given event within a joinpoint context, the composition history does not contain any previous values held by that event. In some cases, it may be possible to calculate previous values by inspection of `#Transform` history records, and reversing the action of the transform. However since not all transformations are reversible, this approach is not always feasible.

If a pointcut needs information relating to previous values held by an event, then it must refer to the hyperspace from which the event was derived. Hyperspace location records within the composition history provide sufficient information to allow programmatic navigation to the correct CMU and CMU index in order to locate previous values.

2.7 Crosscutting Concerns

The examples that we have described in the foregoing sections have demonstrated the application of MusicSpace aspects to simple, single conditions. However, crosscutting concerns, by definition, may be required to be implemented at multiple, possibly otherwise unrelated, locations. Since pointcuts are Smalltalk expressions, any number of conditions, such as those described above, may be logically composed to express crosscutting relationships.

3. Summary

In this tutorial we have described some of the ways in which AspectMusic enables crosscutting concerns to be managed. We have shown that HyperMusic enables musical fragments to be arbitrarily organised into a hyperspace and composed together to form new musical material with reference to the user’s organisational structure. We have described ways in which HyperMusic enables the composer to move freely between musical dimensions, coping with incompleteness and enabling decisions to be deferred. We have also shown the role of hyperslices and regular expressions in defining relationships for which content is not necessarily completely defined. MusicSpace has been described as a temporal framework in which CMUs may be organised in sequence and in parallel to form complete compositions. We have shown that MusicSpace enables orthogonal concern implementations, expressed as MusicSpace aspects, to be associated with a wide range of conditions, and that crosscutting concerns may be described by the logical composition of such conditions. We have briefly outlined a logic representation of MusicSpace joinpoint contexts, and illustrated the use of this representation as a convenient alternative to object representation and as a means by which pointcuts may be expressed declaratively.