AspectMusic/Score

DRAFT 1.0

June 2009

Patrick Hill

1 Introduction

The AspectMusic/J system is a collection of classes which together provide an implementation of Aspect-Oriented Music Representation (AOMR) using the Java programming language. AspectMusic/J provides implementations of both the asymmetric and symmetric AOMR components, respectively as MusicSpace and HyperMusic. Developing a piece of music using AspectMusic/J naturally involves writing Java code.

AspectMusic/Score builds on AspectMusic/J in order to provide a means through which a whole, or partial, music composition, is expressed in terms of AspectMusic structures and operations, represented as an XML document or "score". As such, AspectMusic/Score is an attempt to remove the requirement for AspectMusic/J users to write Java code.

AspectMusic/Score scores are interpreted by a program, called a "Score Processor", which supports key features of both HyperMusic and MusicSpace. In addition, the Score Processor exploits the strategy pattern [1], "plug-in" architecture of AspectMusic/J, enabling AspectMusic/J extensions to be used where applicable.

2 The Score Document

In order to understand the possibilities and limitations of AspectMusic/Score, it is important to first understand the structure of the score document. The score is formally defined by the XML schema `AspectMusicScore.xsd`, which forms part of the AspectMusic/Score package.

The score consists of three key sections, which are described in the following sections.

2.1 HyperMusic Repositories

In AOMR [2], music fragments are expressed as structures called Composed Music Units (CMUs). Central to AOMR is the MDSoC-inspired [3] notion of the "hyperspace" as a container and a organisational system for CMUs. In AspectMusic/J the hyperspace structure is termed a *repository*.

An AspectMusic/Score score file must declare all repositories that are to be used. Each repository is assigned a name through which it will be referred to within the score.

AspectMusic/J supports persistent repositories. Consequently, each repository declaration may include a file name. If the file exists, then the repository is populated from that file.

To enable any modifications made to the repository as part of the score processing to be preserved, each repository may be configured to be written once score processing has completed.

A new, blank repository may be created by specifying the attribute `new="true"`.

The following example shows two repositories being declared. One repository, to be populated from the file "`repo1.xml`", will be referred to as "`temp`". No changes to this repository will be reflected in `repo1.xml`. The second is a new repository which will be referred to as "`main`" and will be persisted to the file "`newrepo.xml`".

```
<repositories>
     <repository filename="repo1.xml"
                     name="temp"
                     write-on-exit="false"
                     zipped="false"
                     new="false"/>
     <repository filename="newrepo.xml"
                     name="main"
                     write-on-exit="false"
                     zipped="false"
                     new="true"/>
 </repositories>
```

2.2 HyperMusic

AspectMusic/J's HyperMusic component is concerned with the construction of CMUs through generative, transformational and combinatorial processes. The CMUs which form the inputs and outputs of these processes are referred to by means of their repository coordinates.

HyperMusic operations are declared between `<hypermusic>` tags within the score file.

2.2.1 Importing Data into CMUs

In order for HyperMusic to be able to combine and transform CMUs, some initial CMU content is required. Of course, CMUs that exist in imported repositories will be imported with their content. However, it is sometimes desirable to be able to construct new CMUs with initial content directly from the score.

AspectMusic/Score enables data to be imported into CMUs using AspectMusic/J's import filter mechanism[1]. Broadly speaking, an import filter is a Java class that populates a CMU. AspectMusic/Score enables import filters to be packaged into a sequence which together construct the desired CMU content.

The following example illustrates an import specification in which a CMU is populated by a single import filter, `PitchSequenceSetterFilter`, and the resultant CMU stored as the unit named "`Melody1`" in the "`Themes`" concern of the "`Section1`" dimension of the repository named "`main`" .

```
<import-specifications>
  <import-specification comment="Melody1">
    <repository-name>main</ns0:repository-name>
      <target-location>Section1;Themes;Melody1</ns0:target-location>
      <import-filters>
        <import-filter
         class="aspectmusicj.score.util.PitchSequenceSetterFilter">
          <filter-parameters>
            <parameter name="pitchSequence"
                       value="A5,C5,A5,G#4,A4"/>
            <parameter name="voice"
                       value="1"/>
            <parameter name="classifier"
                       value="pitch"/>
          <filter-parameters>
      <import-filter>
    <import-filters>
  <import-specification>

  …

</import-specifications>
```

In this example, `PitchSequenceSetterFilter`, which is part of the AspectMusic/Score package, enables a pitch sequence to be set on a particular classifier and voice within a CMU. Pitches are represented using a simple "pitch-class and octave" representation in the parameter named "pitchSequence".

[1] The import mechanism is actually part of the Multi-Dimensional Data Representation (MDDR) upon which AspectMusic/J is built.

AspectMusic/J itself contains filters that enable pitch and rhythm information to be imported from standard MIDI files. Note however that because the AspectMusic/Score processor dynamically loads and instantiates filters, *any* AspectMusic/J filter implementation that is available in the classpath may be used within an AspectMusic/Score score.

While class names, parameter names and parameter values vary between different kinds of import filters, all filters are described within an AspectMusic/Score score file using this pattern.

2.2.2 Symmetric Composition

The main purpose of the symmetric component of AOMR, implemented as AspectMusic/J's "HyperMusic", is to support the composition of evolutionary "chains" [4] of CMUs through transformational and combinatorial operations

AspectMusic/Score supports this "symmetric composition" through composition specifications which are not dissimilar to the Hypermodule Specifications described in [2]. Composition specifications are defined between `<composition-specifications>` tags within the `<hypermusic>` section.

The following example shows a specification for composing the CMU Theme1 into the Themes concern of the Section1 dimension of the repository referred to as "main".

The composition expression requires that all CMUs whose names end in "1" and exist in the Themes concern of any dimension of this repository, subject to the repository slice specification, are to be matched and woven together according to the "mergeByName" composition strategy. The repository slice specification in this example restricts matching to only those CMUs found in any concern within the Section1 dimension.

```
<composition-specifications>
  <composition-specification comment="test">
    <repository-name>main</repository-name>
    <target-location>Section1;Themes;Theme1</target-location>
    <repository-slices>
      <slice>Section1;.*;.*</ns0:slice>
    <repository-slices>
    <composition-expression>
      .*;Themes;.*1
    </composition-expression>
    <composition-strategy>mergeByName</composition-strategy>
  </composition-specification>

  …
```

```
</composition-specifications>
```

Currently, only the standard AspectMusic/J symmetric composer is available within AspectMusic/Score.

The standard composer supports AOMR composition expression syntax. In this scheme, CMUs are represented by three regular expressions separated by ";" representing Dimension, Concern and Unit Name.

Three composition operators are supported. Sequential composition is represented by "+". parallel composition by "|", and evaluation by "@". Parenthesis may be used to indicate evaluation precedence.

Both `mergeByName` and `overrideByName` composition strategies are supported.

3 MusicSpace

AspectMusic/J's MusicSpace component is concerned with the temporal arrangement of the content of CMUs. In addition, the arrangement may be evaluated against a set of "MusicSpace aspects" which may modify the arrangement based upon a set of contextual conditions.

3.1 Defining a MusicSpace

All MusicSpace operations are defined between `<musicspace>` tags. The pulses per quarter note (PPQ) and time signature of the MusicSpace is defined as part of the opening `<musicspace>` tag. By default, these values are set respectively to 480 and 4/4.

```
<musicspace ppq="480" time-signature="4/4">
```

3.1.1 Logic Gateway Definition

The behaviour of MusicSpace aspects is "triggered" when a condition, called a pointcut, is satisfied. The values that can be considered when evaluating the condition are contained in a structure called a *joinpoint context*.

In addition to MusicSpace aspects that use procedural evaluation of joinpoint contexts, AspectMusic/J supports MusicSpace aspects that use declarative, logic-based joinpoint evaluation. This is achieved by causing salient joinpoint information to be reified as logic assertions and for these assertions to be declaratively queried. However, rather than implementing the logic language itself, AspectMusic/J delegates such logic operations to an external logic language implementation through its "logic gateway" abstraction.

Since AspectMusic/Score scores aim to avoid procedural coding, the logic-based approach is used. This requires a logic gateway to be defined within the score. The logic gateway definition specifies the name of a class that implements the logic gateway along with an initialisation parameter.

```
<logic-gateway    class="aspectmusicj.JLogGateway.JLogRepository"
                  init-string="kb.pro"/>
```

3.1.2 MusicSpace Aspect Definition

In order for MusicSpace aspects to be applied to a MusicSpace, they must first be declared. All MusicSpace aspects are declared between the `<aspects>` tags within the `<musicspace>` section of the score.

An aspect declaration consists of naming the class and aspect instance, setting any parameters associated with the aspect, and setting the before and after pointcut expressions.

```
<aspect class="aspectmusicj.demos.StaccatoAspect"  name="Staccato">
  <parameter name="targetParts">
    <list>
      <value>Strings</value>
    </list>
  </parameter>
  <before-pointcut>true</before-pointcut>
  <after-pointcut>true</after-pointcut>
</aspect>
```

Like import filters, aspect instances are dynamically created. Thus, any AspectMusic/J MusicSpace aspect class that is available on the classpath may be used.

3.1.2.1 MusicSpace Aspect Parameter Definition

AspectMusic/Score supports three types of MusicSpace aspect parameter; scalar, list and map.

Each parameter is defined with <parameter> tags, with the name attribute setting the name of the parameter.

Scalar parameters are expressed with a value between `<scalar>` tags.

```
  <parameter name="scalarParam">
    <scalar>ScalarValue</scalar>
  </parameter>
```

List parameters are expressed as `<value>` tags between `<list>` tags.

```
  <parameter name="listParam">
    <list>
```

```
      <value>Value One</value>
      <value>Value Two</value>
      <value>Value Three</value>
    </list>
  </parameter>
```

Map parameters are expressed as `<entry>` tags between `<map>` tags.

```
  <parameter name="mapParam">
    <map>
      <entry key="KeyValOne">Value of KeyValOne</entry>
      <entry key="KeyValTwo">Value of KeyValTwo</entry>
      <entry key="KeyValThree">Value of KeyValThree</entry>
    </map>
  </parameter>
```

In order to set the parameter value on the aspect instance, the method `set*Name* ()` is invoked, where *Name* is name of the parameter, with its first letter capitalized.

1. For example, the parameter named "`mapParam`" is set by invoking the method `setMapParam()`.

3.1.3 MusicSpace Arrangement

A MusicSpace contains the content of selected CMUs arranged in time and across arbitrary parts.

The arrangement of a MusicSpace is defined as a set of <arrangement-item> sections, between <musicspace-arrangements> tags. Each <arrangement-item> selects a CMU from a repository and positions it at a metrical location within a part.

```
  <arrangement-item comment="test1">
    <repository-name>main</repository-name>
    <cmu-location>Section1X;Themes;Theme1</cmu-location>
    <part-name>Strings</part-name>
    <metrical-location>
      <bar>1</bar>
      <beat>1</beat>
      <tick>0</tick>
    </metrical-location>
   </arrangement-item>
```

3.2 MusicSpace Compilation

In order to compile a MusicSpace, a MusicSpace compiler must be defined within the `<musicspace>` section.

The compiler definition names the class that implements the compiler and specifies the resolution of the compilation process, in ticks.

```
<compiler
    class="aspectmusicj.MusicSpace.compilers.MusicSpaceCompiler"
    resolution="15"/>
```

As described in [2], the MusicSpace compilation process involves iterating through the MusicSpace, generating joinpoints and evaluating aspects at each tick. The resolution value is the number of ticks between joinpoints. A value of one generates joinpoints at every tick. Resolution is largely used to help improve compilation speed.

To illustrate resolution, consider the default MusicSpace PPQ of 480. This value enables a range of beat subdivisions to be expressed as integer values. In this scheme, thirty-secondths of a beat (hemi-demi-semiquavers) are represented by the duration value 30. Assuming that this is the smallest rhythmic interval contained by the music, then setting resolution to 30 will result in a significant reduction in compilation time compared to using a resolution of 1.

3.3 MusicSpace Rendering

AOMR and AspectMusic/J are predominantly concerned with the definition and manipulation of data structures that represent music. In order to help maintain generality, AOMR deliberately avoids committing to any particular realisable music representation. Nonetheless, it is important to be able to audition the output of an AspectMusic/J process.

Within AspectMusic/Score, a populated MusicSpace can be rendered to a file by defining a suitable renderer. AspectMusic/J provides a renderer, `MusicSpaceMidiRenderer`, that is capable of transforming a MusicSpace into a standard MIDI sequence. Again, renderer classes are loaded dynamically, so any available renderer may be used.

```
<renderer
  class="aspectmusicj.MidiToolkit.rendering.MusicSpaceMidiRenderer"
  filename="MyMidiFile.mid"/>
```

4 The Score Processor

Now that we have outlined the content of the score document, we can consider the operation of the score processor.

4.1 Running the Score Processor

The score processor is implemented by the class `aspectmusicj.score.ScoreProcessor`. This class implements a `main()` method

taking a single argument containing the name of a score file. Thus it is possible to run the score processor from the command line.

Alternatively, a Java program may construct a `ScoreProcessor` and invoke its `process()` method, passing the name of a score file to be processed.

4.2 Scoring Options

The score processor processes the three sections, <repositories>, <hypermusic> and <musicspace> in order. The <repositories> section is mandatory, however the <hypermusic> and <musicspace> sections are optional. Consequently, it is possible to obtain different behaviours and outputs from the score processor. For example, it is possible to compose CMUs and create or update hyperspaces by working only within a <hypermusic> section. Alternatively, it is possible to just construct and render a MusicSpace by working only with a <musicspace> section.

While the score itself is ostensibly a declarative representation, the score processor reads each section from top to bottom. This explicit ordering is important for establishing the correct sequencing of hypermusic compositions and aspect definition.

4.3 Dynamic Class Loading

As has been mentioned, several objects referred to in the score, such as import filters, logic gateway, MusicSpace aspects and MusicSpace renderers, are loaded dynamically. This enables MusicSpace/Score to operate with implementation objects that were not defined a priori.

In order for these objects to be loaded, their class files must be available in the Java class path that is current when the score processor is invoked. As a minimum, the MDDR, AspectMusic and AspectMusicScore jar files must be in the classpath.

5 Supplied Plug-in Classes

There are a number of classes within the AspectMusic/J distribution which may be used as plug-in classes. This section briefly outlines these classes so that scores that use them can be constructed.

5.1 Import Filters

As has been described, AspectMusic/J Import Filters enable, in various ways, data to be imported into Composed Music Units.

5.1.1 PitchSequenceSetterFilter

The `aspectmusicj.score.util.PitchSequenceSetterFilter` class provides a means of setting any classifier such that it contains pitch values that are embedded within the score.

Parameters

pitchSequence	A sequence of comma-separated pitch values. Each value contains the pitch class and octave of the desired pitch. Simple sharps and flats are supported respectively by the modifier # and b. For example, "A5 G#5 Bb4 G4"
voice	The voice number within the target classifier that should be set with the pitches in the pitchSequence.
classifier	The name of the classifier to be set. Typically, this is "pitch".

5.1.2 RhythmSetterFilter

The `aspectmusicj.score.util.RhythmSequenceSetterFilter` class provides a means of setting any classifier such that it contains rhythm values that are embedded within the score.

Parameters

rhythmSequence	A sequence of space-separated rhythm values. Each value contains the relative onset time and duration, in ticks, of a rhythmic element. For example, "(0,240) (0,240) (0,480)" represents the sequence quaver, quaver, crotchet[2] using the default MusicSpace PPQ of 480 ticks per quarter note.
voice	The voice number within the target classifier that should be set with the pitches in the rhythmSequence.
classifier	The name of the classifier to be set. Typically, this is "rhythm".

[2] Eighth-note, eighth-note, quarter-note

5.1.3 ChordSequenceSetterFilter

The `aspectmusicj.score.util.ChordSequenceSetterFilter` class provides a means of setting any classifier such that it contains chord values that are embedded within the score.

Parameters

chordSequence	A sequence of space-separated chord values. Each chord contains one or more pitch elements, specified as described in 5.1.1. For example, "(F4,A4,C5) (G4,B4,D4,F4) (C4,E4,G4)" represents the chord sequence Fmaj, G7, Cmaj.
classifier	The name of the classifier to be set. By default, this is "harmony".

5.1.4 TransformSetterFilter

The `aspectmusicj.score.util.TransformSetterFilter` class provides a means of constructing a CMU containing a single transformation whose class name is embedded in the score. The transformation is always placed in the transform classifier.

Parameters

transformClass	The fully-qualified name of the transform class. The class must be available on the classpath.

5.1.5 MidiMonoPitchFileImportFilter

The `aspectmusicj.MidiToolkit.importing.MidiMonoPitchFileImportFilter` class provides a means to extract a pitch sequence from a MIDI file and import it into a CMU. Note that `MidiMonoPitchFileImportFilter` always imports to the "pitch" classifier.

Parameters

filename	The absolute path to the file to be imported.
midiTrack	The 1-based number of the MIDI track containing the pitch data to be imported.
midiChannel	The MIDI channel number (0..15) containing the pitch data to be imported.
sequenceName	The value to be stored as the sequence name in the composition history of pitch elements imported from this sequence

5.1.6 MidiRhythmImportFilter

The MdiIrhythmImportFilter class provides a means to extract a simple rhythmic sequence into a CMU from a MIDI file. Note that `MidiRhythmImportFilter` always imports to the "rhythm" classifier.

Parameters

filename	The absolute path to the file to be imported.
midiTrack	The 1-based number of the MIDI track containing the pitch data to be imported.
midiChannel	The MIDI channel number (0..15) containing the pitch data to be imported.
sequenceName	The value to be stored as the sequence name in the composition history of pitch elements imported from this sequence

5.2 Logic Gateways

The `aspectmusicj.JLogGateway.JLogRepository` logic gateway implementation is provided as part of AspectMusic/J. This gateway interfaces to the open-source JLog Prolog implementation [5].

The initialisation parameter (`init-string`) passed to the gateway is the path to a prolog file which contains any additional prolog definitions that are required to process a particular score.

5.3 MusicSpace Compilers

The `aspectmusicj.MusicSpace.compilers.MusicSpaceCompiler` class is the standard MusicSpace compiler provided with AspectMusic/J.

5.4 MusicSpace Renderers

The `aspectmusicj.MidiToolkit.rendering.MusicSpaceMidiRenderer` class is the standard MusicSpace renderer provided with AspectMusic/J. This renderer transforms a MusicSpace into a basic MIDI representation containing pitch, dynamic and rhythmic information.

The extended MIDI renderer, `aspectmusicj.MidiToolkit.rendering.MusicSpaceMidiRendererEx`, is also capable of rendering a chord objects.

5.5 Transformers

5.5.1 TransposeTransform

The `aspectmusicj.HyperMusic.StandardTransformer.TransposeTransform` enables the pitches contained by a given classifier to be transposed by a given number of semitones.

Parameters

delta	The number of semitones to transpose by. The value may be negative.
unitType	The classifier whose content is to be transposed. Typically "pitch"

5.5.2 VoiceReverser

The `aspectmusicj.HyperMusic.StandardTransformer.VoiceReverser` reverses the index order of all the primitives for each classifier for a given voice number.

Parameters

voice	The voice to be reversed

6 A Worked Example

In this section we briefly describe the worked example given in the file BachMelody.xml.

In overview, this example attempts to build the first eight bars of the Violin I part of Bach's Harpsichord Concerto Number 5 (BWV 1056).

6.1 Musical Structure

In order to construct the first eight bars of the Concerto, the structure of these bars has been analysed and repeating fragments of melody and rhythm identified. This decomposition into rhythmic and melodic "building blocks" is rather simplistic. However, it serves the purpose of this example in demonstrating some of the features of AspectMusic/Score.

In overview the first eight bars can be considered as an *almost* exactly repeated four-bar structure. The repeating structure consists of three "question-answer" phrases, the last of which receives a different one-beat "coda" for each instance. In addition, the second instance of the four-bar structure is transposed up a perfect fourth.

6.1.1 Pitch Decomposition

The decomposition considering only the pitch dimension yields three different "question" fragments (Q1, Q2, Q3), two "answer" fragments (A1,A2) and two "codas" (C1, C2).

6.1.2 Rhythm Decomposition

The decomposition considering only the rhythm dimension yields three "question" fragments (Q1,Q2,Q3), and one "answer" fragment A1. The rhythm of the codas will, for this example, be considered to be related to the rhythm fragments already created for the question fragments. As such, no new rhythmic fragments are required for the codas *in this analysis*.

6.2 Phrases

We use the term "phrase" to denote a structure that is populated with both pitch and rhythm information. In this example, the pitch and rhythm fragments align exactly.

6.3 Schema Location

The root of the score document declares that the document conforms to the XML schema "AspectMusicScore.xsd", and provides a path to the schema file and assigns the namespace prefix "ams" to elements from that schema.

```
<ams:aspectmusic-score
xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
xmlns:ams="http://xml.netbeans.org/schema/AspectMusicScore"
xsi:schemaLocation="http://xml.netbeans.org/schema/AspectMusicSc
ore xsd/AspectMusicScore.xsd" name="">
```

6.4 Repository Declaration

The example creates a single repository which is referred to as "main", and which will be persisted to the file "repo/BachBWV1056-1.xml".

```
<ams:repositories>
      <ams:repository filename="repo/BachBWV1056-1.xml"
                  name="main"
                  write-on-exit="true"
                  new="true"
                  zipped="false"/>
</ams:repositories>
```

6.5 Import Specifications

The example defines musical material within the score using the `RhythmSequenceSetterFilter` and `PitchSequenceSetterFilter` import filters described in section 5 to populate CMUs with the rhythmic and melodic fragments outlined in section 6.1.

Additionally the TransformSetterFilter is used to make available the TransposeTransform transformer.

6.6 Composition Specifications

The composition specifications define the structural relationships between various fragments in repository.

There are two main types of composition specification in this example. Firstly, there are those that bring together pitch and rhythm dimensions to form "phrases".

The following example constructs the "question 1" phrase by selecting an merging those CMUs called "Q1" from the Motives;Rhythm and Motives;Melody repository slices.

```
<ams:composition-specification comment="Question 1">
      <ams:repository-name>main</ams:repository-name>
      <ams:target-location>Themes;Phrases;Q1</ams:target-location>
      <ams:repository-slices>
            <ams:slice>Motives;Rhythm;.*</ams:slice>
            <ams:slice>Motives;Melody;.*</ams:slice>

      </ams:repository-slices>
      <ams:composition-expression>.*;.*;Q1</ams:composition-
expression>
      <ams:composition-strategy>mergeByName</ams:composition-
strategy>
</ams:composition-specification>
```

Secondly, there are composition expressions that construct new structures through sequential composition or transformation of other structures. The following example specifies a structure QA3 which the phrase Q3 sequentially composed with the phrase A2.

```
<ams:composition-specification comment="QuestionAnswer3">
      <ams:repository-name>main</ams:repository-name>
      <ams:target-location>Themes;Phrases;QA3</ams:target-location>
      <ams:repository-slices>
            <ams:slice>Themes;Phrases;.*</ams:slice>
      </ams:repository-slices>
      <ams:composition-expression>.*;.*;Q3                          +
.*;.*;A2</ams:composition-expression>
      <ams:composition-strategy>mergeByName</ams:composition-
strategy>
</ams:composition-specification>
```

The following example shows the production of a new CMU by transforming an existing CMU, in this case by transposition.

```
      <ams:composition-specification comment="IntroPhraseTransposed">
                        <ams:repository-name>main</ams:repository-
name>
                        <ams:target-
location>Themes;Phrases;IntroTrans5</ams:target-location>
                        <ams:repository-slices>

      <ams:slice>Themes;Phrases;.*</ams:slice>

      <ams:slice>Transform;Pitch;.*</ams:slice>
                        </ams:repository-slices>
                        <ams:composition-expression>@(.*;.*;Intro2  +
.*;.*;Transpose[delta=5,unitType=pitch])</ams:composition-expression>
                        <ams:composition-
strategy>mergeByName</ams:composition-strategy>
                  </ams:composition-specification>
```

6.7 Arranging CMUs in MusicSpace

In this example, we want to arrange two CMUs in MusicSpace. We then want to manipulate the data in the MusicSpace to add some articulation details, using logic-based MusicSpace aspects.

First we declare the MusicSpace and its parameters.

```
<ams:musicspace ppq="480" time-signature="2/4" tempo="65">
```

Since we want to use logic-based aspects, the MusicSpace must be associated with an AspectMusic/J Logic Gateway implementation. In this example, we use the JLog gateway. The JLog gateway implementations takes an initialisation parameter that names a file of PROLOG statements to be pre-loaded into the repository.

```
<ams:logic-gateway
      class="aspectmusicj.JLogGateway.JLogRepository"
      init-string="kb.pro"/>
```

Following the aspect defintions, which we will discuss in section 6.9, the musicspace-arrangements section simply contains two arrangement-items, one each for the original and transposed version of the melody contained respectively in the CMUs Themes;Phrases;Intro1 and Themes;Phrases;IntroTrans5.

6.8 Evaluating and Rendering the MusicSpace

MusicSpace evaluation and rendering is coordinated by the Score Processor using plug-in implementations. The Score defines the implementations to use and passes any required parameters to these implementations.

In this case, we want to evaluate the MusicSpace using the provided MusicSpaceCompiler, with a resolution of 10 ticks.

Following evaluation, we want to use MusicSpaceMidiRendererEx to render the MusicSpace as a MIDI file, named output/BachMelody.mid.

```
<ams:compiler
class="aspectmusicj.MusicSpace.compilers.MusicSpaceCompiler"
resolution="10"/>

<ams:renderer
class="aspectmusicj.MidiToolkit.rendering.MusicSpaceMidiRendererEx"
filename="output/BachMelody.mid"/>
```

6.9 Aspect Definitions

In this example, we would like to make certain notes “staccato”. The aspectmusicj.demos package provides a simple “Staccato” aspect

implementation. All that is required to apply this aspect is to determine appropriate before pointcut expressions.

We shall illustrate three pointcut expressions that demonstrate some of the different kinds of conditions that can be expressed.

6.9.1 Metrical Location

One way to select notes of interest is to specify the metrical location at which they occur.

The following pointcut selects any notes whose onset is on or within beat 2 of any bar Note that a cuepoint/2 fact is generated for each part in the MusicSpace. This is because, in principle, AspectMusic/J permits each part to have its own time signature. Therefore different parts may have different metrical locations for the same joinpoint. However, the current AspectMusic/Score implementation supports only a global time signature within a MusicSpace. Consequently, the part name, which is the first parameter of the cuepoint predicate, is unimportant.

```
<ams:before-pointcut>
   cuepoint(_, location(_,2,_))
</ams:before-pointcut>
```

6.9.2 Transformation

Another method of selecting notes of interest is if they have been transformed in some way.

The following pointcut selects any notes that have been transformed by the TransposeTransform at any point in their history.

```
<ams:before-pointcut>
   history(strings,pitch,_,_,H),
   =(H,transform(P)),
   member(param(class,"aspectmusicj.HyperMusic.StandardTransformer
   .TransposeTransform"),P)
</ams:before-pointcut>
```

6.9.3 Repository Location

The final method of selecting notes that we will consider here is by repository location. The following pointcut selects those notes that are currently located in a CMU called “IntroTrans5”.

This expression excludes notes that have been derived from IntroTrans5 by considering only the most recent repositoryLocation history event. This is

achieved by ensuring that the historyIndex parameter (I) of the selected history event is equal to or greater than any other historyIndex that relates to a repositoryLocation event.

For clarity, this expression is rather underconstrained; we have not, for example, specified dimension and concern.

```
<ams:before-pointcut>
   history(strings,pitch,_,I,H),
   history(strings,pitch,_,L,M),
   >=(I,L),
   =(H,repositoryLocation(P)),
   =(M,repositoryLocation(_)),
   member(param(unit,"IntroTrans5"),P)
</ams:before-pointcut>
```

7 Summary

In this document we have described AspectMusic/Score as a means of using AOMR through AspectMusic/J without necessarily writing Java code. Rather, musical scores are expressed using an XML representation, and processed by a Score Processor.

While AspectMusic/Score is naturally a more limited that a general purpose programming system, as we have described, it is possible to develop AspectMusic/J extensions and use them within the score language.

We have described a worked example, which although producing rather trival musical output, demonstrates some of the key features of AspectMusic/Score.

8 References

[1] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns. Elements of Reusable Object-Oriented Software*: Addison Wesley, 1996.

[2] P. Hill, S. Holland, and R. Laney, "An Introduction to Aspect Oriented Music Representation," *Computer Music Journal*, vol. 31, pp. 47-58, 2007.

[3] H. Ossher and P. Tarr, "Multi-Dimensional Separation of Concerns in Hyperspace," IBM T.J.Watson Research Center RC 21452(96717)16APR99, 1999.

[4] M. Crochemore, C. S. Iliopoulos, and Y. J. Pinzon, "Computing Evolutionary Chains in Musical Sequences," *The Electronic Journal of Combinatorics*, vol. 8, 2001.

[5] "JLog - Prolog in Java," 1.3.6 ed, 2007.