An AspectMusic/J – Tutorial

DRAFT 1.0

June 2009

Patrick Hill

1 Introduction

The composition of music in many idioms involves the exploitation of recurrent, recombinant musical fragments. Any given fragment may, as a consequence, appear in arbitrarily many structures, in its original or transformed state. Such a fragment is said to *crosscut* the musical structure, in the sense that the modification of such a fragment implies that revisions should be made to related structures.

Aspect-Oriented Music Representation (AOMR), is an approach to music representation that, drawing inspiration from Aspect-Oriented Programming (AOP) and cognate techniques in computer software, aims to reduce various kinds of crosscutting in music. AOMR consists of two related representations. The *symmetric* representation, in overview, enables fragments of music to be encapsulated and associated with user-defined areas of compositional interest. New fragments may be generated by specifying transformational and combinatorial relationships with other fragments, by reference to their area of interest. In this way, AOMR separates structure from content, and enables crosscutting fragments to be stated once, with any subsequent revisions to a fragment being automatically propagated to all related fragments.

In order to remain recombinant, each fragment must be independent of its ultimate temporal location. The *asymmetric* representation of AOMR provides an approach to the arrangement of fragments within a temporal framework, and enables the content of fragments to be conditionally modified, based on factors such as location, context and provenance.

AspectMusic/J is an implementation of Aspect-Oriented Music Representation (AOMR) [1, 2] written in the Java language. In this implementation the symmetric and asymmetric AOMR components are respectively named HyperMusic and MusicSpace.

An important feature of the asymmetric component of AOMR is its provision for declarative pointcut expressions. AspectMusic/J uses a strategy pattern [3] architecture to enable different logic language implementations to be "plugged-in".

While AOMR itself prefers no particular concrete music representation, AspectMusic/J is aimed at MIDI representations. As such, its default

representation of musical primitives, and accompanying tools support the production of, and extraction of data from, MIDI sequences.

This tutorial is a non-exhaustive guide to AOMR and AspectMusic/J. Upon completing the tutorial, the reader should be in a position to experiment with AspectMusic/J. However, the references, JavaDocs and the AspectMusic/J source code, particularly the demonstration package, should be consulted for further detailed information.

2 HyperMusic

Symmetric AOMR and its realisation in the HyperMusic system, as described in [2], supports the separation and composition of multi-dimensional musical concerns. The principal constructs of HyperMusic, such as the Composed Music Unit, Hyperspace and Hypermodules described below, may be expressed as specialisations of more general structures. As a consequence, the implementation of these constructs has been abstracted out to form Multi-Dimensional Data Representation (MDDR) in the package `mddr`. In AspectMusic/J, the HyperMusic system is an extension of MDDR and is implemented in the package `aspectmusicj.HyperMusic`.

2.1 Composed Music Units

The Symmetric component of AOMR, as explained in [1, 2], is concerned with the construction of reusable music fragments which are expressed as Composed Music Units (CMUs).

CMUs are implemented by the AspectMusic/J class `ComposedMusicUnit`, which extends the MDDR class `mddr.ComposedDataUnit`.

The structure of a CMU is effectively the same as described in [1, 2]. However there is a slight change in nomenclature that, it is hoped, clarifies the purpose of the various components of a CMU.

In AspectMusic/J, a CMU contains one or more Voiced Music Units (VMU), each associated with a classifier name that broadly specifies the kind of information that the VMU contains. VMUs are implemented by the class `VoicedMusicUnit`. which maps a voice identifier (integer) to a Single-Voiced Music Unit (MU) implemented by `MusicUnit`.

Each MU is a collection of `MusicUnitItem` objects, which are, in turn, a thin wrapper around the MDDR class `DataItem`. `DataItem`, and therefore `MusicUnitItem`, instances encapsulate an arbitrary object with a composition history. AspectMusic/Js implementation of Composition History is described in Section 4.3

2.2 Hyperspace

Drawing inspiration from MDSoC [4], in Symmetric AOMR, each CMU is contained by a structure termed a *Hyperspace.* The key purpose of the hyperspace is to provide a coordinate system, consisting of the triple Dimension, Concern and Unit name, by which each contained CMU is uniquely addressed.

Dimensions, concerns and unit names are arbitrary string values. Thus AOMR users are able to organise CMUs within the hyperspace according dimensions, concerns and names that are meaningful to them.

In AspectMusic/J, the hyperspace is implemented by the MDDR class `Repository`.

2.2.1 Hyperspace Persistence

The ability to persist a hyperspace, and its contents, provides a means through which musical ideas expressed using the AOMR formalism may be saved, versioned and shared.

In our original (Smalltalk) AspectMusic implementation, a hyperspace and its content can be persisted between interactive sessions in two ways. Firstly, any existing hyperspace would naturally be persisted as part of a Smalltalk image. Secondly, the Smalltalk implementation of Hyperspace includes methods that enable hyperspace instances to be written to and read from a separate file using the Binary Object Storage System (BOSS).

Java does not benefit from the concept of an image. However, AspectMusic/J does provide the ability to save and load hyperspaces.

Hyperspace persistence is achieved using Java's `XMLEncoder` and `XMLDecoder` to write and read MDDR Repository structure and content as XML data.

Any `Repository` is capable of being saved and loaded so long as its content conforms to Java's XML encoding rules or a suitable encoding delegate is provided. The `HyperMusicRepositoryPersistenceHelper` provides an easy way of reading and writing hyperspace instances. This class automatically configures the required persistence delegates for all the standard AspectMusic/J primitives. In addition, the helper is also capable of transparently creating and using compressed XML repository images.

It is possible to extend AspectMusic/J to support persistence for other primitive types.

2.3 Composition Specifications

A composition specification (formerly hypermodule specification) describes how a new CMU is to be produced through composition and transformation of CMUs in a Repository. A composition specification, consists of five parts, described below.

2.3.1 Repository

The repository specifies a repository instance from which CMUs will be drawn and into which the resultant CMU will be placed.

2.3.2 Module Location

The Module Location specifies the Repository coordinates of the resultant CMU.

2.3.3 Composition Expression Tree

The composition expression tree value is the root node of a tree that specifies the composition operation to be performed. AspectMusic/J provides an expression evaluator that produces an appropriate expression tree given a String representation of the desired composition expression.

The default expression language enables CMUs to be referenced by Repository location using semicolon-separated regular expressions for Dimension, Concern and Unit Name. This approach supports both extensional and intensional specification of CMUs.

By default, the binary sequential and parallel composition operations are represented, respectively, by the symbols + and |. The unary evaluation operator is represented by a prefixed @. Square brackets may be used to enclose parameters, expressed as key-value pairs. Parentheses may be used to express evaluation precedence.

For example, the following composition expression represents the evaluation of a CMU which results from the sequential composition of the CMU stored a location Rhythms;Motive;Motive1 with that stored at Transformations;General;Retrograde. The single parameter "voice" is passed, with the value "1".

```
@(Rhythms;Motive;Motive1 +
Transformations;General;Retrograde[voice=1])
```

2.3.4 Composition Strategy

As has been noted above, CMUs that are to participate in a composition expression may be specified extensionally or intensionally.

An extensional specification specifies hyperspace locations without using regular expressions. As a consequence only those CMUs that exist at the exact locations specified will be selected.

An intensional specification, in contrast, uses at least one regular expression within a CMU location. As a consequence, the selection of CMUs to be composed will depend on the repository content at the time the composition is executed. It is possible, and probably desirable, that the Repository may contain multiple matches for an intensionally specified CMU. A composition strategy specifies what action should be taken in order to resolve multiple matches.

Two composition strategy implementations are supplied with AspectMusic/J. These are "mergeByName" and "overrideByName". The mergeByName composition strategy sequentially composes, in ascending address order, all those CMUs that match the given repository address. The overrideByName strategy selects only the last matching CMU, again ordered by repository address.

2.3.5 Hyperslices

It may be that only a subset of the content of the Repository should be considered when evaluating a composition expression. A hypermodule's hyperslice specification specifies the Dimensions, Concerns and Unit name patterns that should be considered when selecting CMUs from the Repository.

2.4 The CMU Transformation Framework

As we have described, the ability to derive CMUs from other CMUs is fundamental to the symmetric component of AOMR. One way in which CMUs may be derived is through combinatorial operations such as sequence and parallel. Another important method is transformation.

In AOMR, transformation is a two phase process. In the first phase, transformation implementations are (sequentially) composed into a special "transform" classifier within a CMU. In the second phase, the CMU is "evaluated". Evaluation of a CMU causes all of the transformations in its

transform classifier to be applied, producing a CMU with no transformations in its transform classifier. The scope of a transformation is, therefore, the CMU in which it is evaluated. The separation of composition and evaluation stages means that transformations are "first-class" citizens of the AOMR world, being conceptually equal to music primitives such as pitch, rhythm and dynamic.

The transform classifier name is given by the constant `ComposedMusicUnit.TRANSFORM_CLASSIFIER`.

2.4.1 The Transformation Process

In order to transform a CMU, one or more transformations must be composed into the CMU and then the CMU must be evaluated. Both the composition and evaluation phases are, therefore, specified by composition expressions.

Transformations always compose in sequence. The default parallel and sequential composition operators provided by AspectMusic/J implement the required behaviour.

2.4.2 Parameters and Parameter Injection

While certain kinds of transformation do not require it, it is desirable to be able to define transformations whose operation may parameterised. A "pitch transpose" transformation, for example, might take a parameter indicating the number of semitones by which each pitch is to be transposed.

It is important to understand that in AOMR transformations are not executed directly. Rather, they are implicitly executed as part of the evaluation of a CMU that contains them.

To illustrate this, consider the following example.

```
@(Rhythms;Motive;Motive1 +
Transformations;General;Transpose[delta=5])
```

This composition expression specifies that the CMU "Motive1" is to be sequentially composed with the CMU "Transpose" and then evaluated. The "Transpose" CMU is given the parameter "delta" with the value "5".

When the composition strategy implementation resolves a CMU specification to one or more CMUs, it injects any specified parameters into those CMUs. The CMUs, in turn, pass the parameter values to the transformations held in their transform classifier.

The evaluation process then executes each transformation contained by the transform classifier of the evaluated CMU, returning a new CMU.

2.5 Composition History

Symmetric AOMR is concerned with the evolution of musical fragments from other fragments through combination and transformation. Composition History is a feature of AOMR which enables the derivation of any CMU element to be obtained.

Composition History is implemented by the MDDR classes `mddr.CompositionHistory`, `mddr.CompositionHistoryItem` and `mddr.CompositionHistoryItemFactory`. A `CompositionHistory` is simply a list of `CompositionHistoryItems`. As described above, a `DataItem` wraps a `CompositionHistory` instance with the object to which the history applies.

MDDR implements two kinds of composition history item, namely *Repository Location* and *Transformation History*.

Repository Location items record the location of a `DataItem` when its containing CMU is added to a repository. By default, `Repository` adds a Repository Location history item to the history of each `DataItem` contained by a CMU whenever the CMU is added to the repository.

Transformations may add composition history items to transformed elements. The `CompositionHistoryItemFactory` can provide instances of suitable `CompositionHistoryItems` for classes derived from MDDR's `AbstractCDUTransformation` class. However, it is the responsibility of the derived class to add these history items to those items that it transforms.

2.6 Working with HyperMusic

2.6.1.1 Create a Hyperspace

The main focus of work within HyperMusic is the hyperspace. A hyperspace is created simply by invoking its constructor. If you wish, you can name the hyperspace.

```
Repository hyperspace = new Repository();
hyperspace.setName("MyHyperspace");
```

2.6.1.2 Populate CMUs

In general, work with HyperMusic involves creating new CMUs by combining and transforming CMUs already in the hyperspace. Clearly, however, it is necessary to provide an initial CMU population to work with.

The following code fragment shows a CMU being populated with a simple melody.

```
ComposedMusicUnit tuneCMU = new ComposedMusicUnit();
String[] tune = new String[]{"A4", "B4", "C#5", "G#4", "A4"};
for (String pitch : tune) {
      tuneCMU.addItem(
            ComposedMusicUnit.PITCH_CLASSIFIER,
            1,
            Pitch.getPitch(pitch));
}
```

The CMU may then be added to hyperspace at the required address.

```
hyperspace.putUnit("Melody", "Motive", "Motive1", tuneCMU);
```

2.6.1.3 Initialise the HyperMusic Compiler

Before composition specifications can be compiled to produce new CMUs, the compiler environment must be initialised. AspectMusic/J provides convenient factory methods to initialise a default environment.

```
HyperMusicComposer composer =
      ComposerFactory.getStandardComposer();
```

The default composer automatically registers the default implementations of overrideByName and mergeByName composition strategies, respectively against the names "`overrideByName`" and "`mergeByName`".

```
CompositionExpressionParser parser =
      CompositionParserFactory.getStandardParser();
```

The default expression parser automatically registers the default sequential composition, parallel composition and evaluation operations, respectively against the operators + , |and @. The default expression parser expects parameters to be expressed as key=value pairs, enclosed by square brackets. Parenthesis may be used to indicate precedence.

2.6.1.4 Compose New CMUs

As we have described, new CMUs are constructed by compiling composition specifications. The following fragment shows how the five elements of a CompositionSpecification, described in section 2.3, are populated.

```
CompositionSpecification cspec = new CompositionSpecification();
cspec.setCompositionStrategy("mergeByName");
cspec.setRepository(hyperspace);
cspec.setModuleLocation("Themes", "Melody", "Melody1");
cspec.addRepositorySlice(new RepositoryLocation(".*", ".*", ".*"));
cspec.setExpressionTree(parser.parse(".*;.*;Motive1"));
```

This example creates a CMU stored at the location "`Themes;Melody;Melody1`". The expression "`.*;.*;Motive1`" matches all CMUs named `Motive1`, irrespective of their concern and dimension. The `mergeByName` composition strategy specifies that all matched CMUs will be sequentially composed. The repository slice specification includes the whole repository. Consequently, the CMU created by this composition specification will contain all CMUs named "`Motive1`", sequentially composed.

To compose the `CompositionSpecification`, call

```
composer.compose(cspec);
```

Working with HyperMusic typically involves creating and executing a sequence of `CompositionSpecifications`. The `CompositionSpecificationPackage` class, which is a thin wrapper around `LinkedList<CompositionSpecification>`, enables a sequence of `CompositionSpecifications` to be created. The HyperMusicComposer's compose method is defined for `CompositionSpecificationPackage`, enabling an entire package to be compiled in one go.

2.6.2 Saving and Loading Repositories

AspectMusic/J is capable of saving repositories to an XML representation, and creating populated repositories from this representation. A helper class (`HyperMusicRepositoryPersistenceHelper`) is provided to simplify the processes involved. The XML representation may be written as a text file, or compressed to a ZIP file. The helper contains methods for both saving and loading uncompressed and compressed formats.

```
HyperMusicRepositoryPersistenceHelper persistenceHelper =
new HyperMusicRepositoryPersistenceHelper();

      // To save hyperspace to compressed XML file
persistenceHelper.saveZip(hyperspace, "Hyperspace.xml");
// To load hyperspace from compressed XML file
hyperspace = persistenceHelper.loadZip("Hyperspace.xml");
```

3 MusicSpace

Much of the re-usable nature of CMUs derives from the fact that CMUs are, like Music Structures [5], defined independently of their final temporal locations, if indeed they have any. Of course, to form part of a given musical composition, the content of selected CMU instances must be associated with particular temporal locations.

As described in [1, 2], AOMR's asymmetric component, which is influenced by asymmetric approaches to Aspect Oriented Programming, provides a

framework in which CMU content may be arranged in time. This framework additionally provides a means through which CMU content may be varied, in arbitrary ways, and in response to arbitrary contextual conditions. These conditions and variation implementations are encapsulated as Aspects which are applied through a process of compilation.

MusicSpace is AspectMusic/J's implementation of asymmetric AOMR.

3.1 MusicSpace Aspects

A MusicSpace aspect is a Java class that encapsulates some behaviour, termed *advice*, with a description of those conditions that must be satisfied in order for the advice to be invoked, termed a *pointcut*. MusicSpace aspects have both before- and after- advice respectively associated with before- and after- pointcuts. The purpose of before- and after- pointcut evaluation is described in section 3.2.

In addition to pure Java aspects, AspectMusic/J supports MusicSpace aspects that use a logic language. In this configuration, aspects are able to evaluate declaratively specified pointcuts, expressed as logic queries. The results of such queries are made available to advice implementations.

3.2 MusicSpace Compilation

In MusicSpace, a populated `aspectj.MusicSpace.MusicSpace` instance is compiled against a set of MusicSpace aspects in order to produce a new `MusicSpace` instance. The set of aspects to be considered in a compilation is managed by an AspectManager instance, implemented by `aspectmusicj.MusicSpace.AspectManager`.

In AOMR, as described in [1, 2], the compilation of a `MusicSpace` takes place using a single pass through the `MusicSpace` content. The compilation process involves iterating through the tick positions of the `MusicSpace` and invoking registered aspects passing a joinpoint context object, containing the tick position and details of those events, if any, that start at that position.

The before-pointcut of each aspect is evaluated and, if satisfied, the before-advice is executed. Before advice may modify the joinpoint context which, when the before-point of every aspect has been evaluated, is written to the output MusicSpace instance. The (possibly modified) joinpoint context is then passed to the after-pointcut of each aspect. If an aspect's after-pointcut is satisfied, then its after-advice is invoked. After-advice may not modify the output MusicSpace instance, but may be used to modify the state of the aspect or the aspect manager.

3.2.1 Multi-Pass Compilation

One problem with a single-pass compilation approach is that the resultant `MusicSpace` may not be consistent with the set of aspects that produced it. This situation arises if an aspect modifies the content of the MusicSpace prior to its joinpoint since the single pass compilation will, by definition, never re-evaluate the modified joinpoints.

At first sight, it seems that it is possible to simply re-evaluate those joinpoints that are changed by an advice. However this approach does not account for aspects that consider events at those modified joinpoints but are not themselves evaluated at those joinpoints. For example, an aspect A at a joinpoint J may consider events in the MusicSpace within a window of T ticks in advance of J. Changing one or more events within the window would not cause J to be re-evaluated. Of course the situation becomes more complex when aspects dynamically choose the joinpoints of interest.

The solution chosen for AspectMusic/J is to implement a multi-pass scheme. Using this approach a MusicSpace instance is compiled to produce a resultant MusicSpace. The result is then compiled against the same set of aspects. The process is repeated until either no advice are executed or the number of passes exceeds some user-defined value.

However, a new class of MusicSpace aspects arises as a consequence of this approach. In particular, it is desirable within a compilation to apply certain aspect types at most once at any given joinpoint irrespective of the number of passes within the compilation cycle.

3.3 Working with MusicSpace

3.3.1 Create MusicSpace Aspects

MusicSpace aspects are derived from the class `aspectmusicj.MusicSpace.AbstractAspect`.

This class requires subclasses to provide implementations of `beforePointcut()`, `beforeAdvice()`, `afterPointcut()` and `afterAdvice()` methods.

If the aspect is required to use logic-based pointcut expressions, then the aspect implementation should be based on `aspectmusicj.MusicSpace.AbstractLogicAspect` which requires only `beforeAdvice()` and `afterAdvice()` implementations.

MusicSpace aspects must be annotated `@MusicSpaceAspect`. By default, all MusicSpace aspects so annotated are considered not to be repeatable instances. In other words, the aspect's before-advice will be invoked at most once per joinpoint. If the before-advice is to be invoked as many times as the before-pointcut is satisfied, then it must be annotated as `@MusicSpaceAspect(repeatableInstance = true)`

In the current AspectMusic/J implementation, it is not possible to automatically determine if an aspect's before-advice has modified the joinpoint context. Consequently, the advice must notify the MusicSpace system itself. This process has been made as simple as possible, requiring only that the advice calls `markMusicSpaceAsModified()` on the joinpoint context.

3.3.2 Create a Logic Repository

In order to use aspects that use logic-based pointcut expressions, a logic repository must first be created. AspectMusic/J provides a general interface `ILogicRepository`, described in section 5.2, that can be used to connect AspectMusic/J to a variety of logic language implementations. A default implementation, that works with the open source JLog[1] Java/Prolog implementation is provided by the class `JLogRepository`.

The `JLogRepository` instance must be initialised with a knowledge base that defines the predicates that your aspects will need. In the following example, the knowledge base is provided in text file "kb.pro".

```
ILogicRepository logicRepos = new JLogRepository();
logicRepos.initialise("kb.pro");
```

3.3.3 Create a MusicSpace Structure

A MusicSpace structure consists of a `MusicSpace` instance containing one or more named *parts*, each represented by a `MusicSpacePart` instance. A `MusicSpace` is created by calling its constructor and, if required, passing an initialised `ILogicRepository` instance.

```
MusicSpace musicSpace = new MusicSpace(logicRepos);
```

`MusicSpacePart` instances may be managed by the `MusicSpace` itself. The simplest way to obtain a `MusicSpacePart` instance is to call `getOrCreatePart()` as illustrated below.

```
MusicSpacePart part1 = musicSpace.getOrCreatePart("part1");
```

[1] http://jlogic.sourceforge.net/

By default, `MusicSpaceParts` default to a "common time" time signature. However, a different time signature may be set.

```
part1.setTimeSignature(TimeSignature.sixEight());
```

3.3.4 Populate Parts with CMUs

Once a part has been created it can be populated with the contents of a CMU using the `putCmuEvents` method. The following example shows how to retrieve a CMU from a repository, and place its contents into a `MusicSpacePart` starting from the first beat of the first bar.

```
ComposedMusicUnit myCMU = (ComposedMusicUnit)
      hyperspace.getUnit(
      new RepositoryLocation("Themes", "Melody", "Melody4"));
      part1.putCmuEvents(1,1,0,myCMU);
```

3.3.5 Instantiate and Initialise Aspects

Aspects must be instantiated and initialised prior to `MusicSpace` compilation. The exact process will depend upon the aspect implementations being used.

The following code fragment shows an instance of `aspectmusicj.demos.StaccatoAspect` being created and initialised. In this example, `StaccatoAspect` is a logic-based aspect, derived from `aspectmusicj.MusicSpace.AbstractLogicAspect`. Its before-pointcut is set using the `setBeforePointcutQuery` method. The before-pointcut will be satisfied if the joinpoint context contains an event in the `MusicSpacePart` called "part1" where the pitch dimension of the event is derived from a CMU called "Melody1". The logic representation of joinpoints is discussed in more detail in section 4.4.

```
StaccatoAspect staccato = new StaccatoAspect();
HashSet partSet = new HashSet();
partSet.add("part1");
staccato.setTargetParts(partSet);

staccato.setBeforePointcutQuery("history(part1,pitch,_,_,repositoryLo
cation(X)), member(param(unit,\"Melody1\"),X)");
```

3.3.6 Register Aspects

Before a MusicSpace can be evaluated, an AspectManager must be created, and all the MusicSpace aspect instances that are required to be evaluated must be registered with it. Each aspect instance is named.

```
AspectManager am = new AspectManager();
am.registerAspect("Staccato", staccato);
```

3.3.7 Compiling the MusicSpace

A MusicSpace is compiled by an instance of `MusicSpaceCompiler`, as follows.

```
MusicSpaceCompiler compiler = new MusicSpaceCompiler();
MusicSpace newMusicSpace =
       compiler.compileWithAspectManager(musicSpace,
       am,
       12000,
       15);
```

Note that since it is not possible to be sure when compilation ends, the `compileWithAspectManager` method requires the maximum number of ticks to be specified, in this case 12000. If the value "-1" is specified for the maximum number of ticks then three times the number of ticks in the MusicSpace being compiled will be evaluated.

In addition, the tick resolution may be specified. In this example, ticks will be generated in steps of 15 up to a maximum of 12000.

Compilation will continue until no further modifications are made by any aspect, or until the maximum number of iterations is reached. By default, the maximum number of iterations is five. This value can be modified using the MusicSpaceCompiler's `setMaxIterations` method.

The result of the compilation is a new MusicSpace instance.

4 The Default AspectMusic/J Implementation of AOMR.

As has been explained, AspectMusic/J is both a Java framework for AOMR and an AOMR implementation. While AOMR itself prefers no particular concrete music representation, the default AOMR implementation provided by AspectMusic/J, in terms of the default primitives and supporting tools, is aimed at a MIDI environment.

4.1 Music Representation Objects

AspectMusic/J provides four kinds of music representation objects; Pitch, Chord, Rhythm and DynamicLevel.

Pitch, Rhythm and DynamicLevel primitives are created by factory methods. For any given value, only one instance is created. Instances are immutable. Mutator methods delegate to the factory to return instances that represent a suitably mutated object.

4.1.1 Pitch

Pitch is represented by the Pitch class. Pitch instances are created on demand through the (static) factory method getPitch().

4.1.2 Rhythm

Rhythm, in terms of onset and duration, is represented by the Rhythm class. Rhythm instances are created through the factory method getRhythm().

4.1.3 DynamicLevel

The DynamicLevel class represents note loudness. Instances of DynamicLevel are obtained through the factory method getDynamicLevel().

4.1.4 Chord

The Chord abstraction aggregates Pitch objects in order to provide a convenient solution to the problem of voiced vs. unvoiced representations, which we shall describe below.

The interpretation of the CMU structure of an earlier version of AOMR [6] was such that a CMU could represent only a homorhythmic fragment. Under this interpretation, a rhythm classifier was single-voiced, while a pitch classifier was permitted to contain multiple voices. The interpretation associated each rhythm value with all the pitch elements at the corresponding index.

The limitations which such an interpretation place on the representational capabilities of a CMU are quite severe. Consequently, the approach was revised to the current approach in which the CMU can represent multiple, rhythmically independent voices.

Any homorhythmic CMU can be represented using this new approach. However, a consequence of the approach is that it requires the user to specify rhythmic patterns "horizontally", in terms of discrete monophonic voices, rather than "vertically" in terms of note clusters following a single rhythmic pattern. When dealing with irregularly voiced or unvoiced music, the requirement to specify the rhythmic characteristics of each note of each chord, can become somewhat cumbersome.

The Chord class offers a solution to the specification of homorhythmic structures. A Chord instance, as might be expected, represents an ordered collection of Pitch objects. Crucially, however, a Chord instance can be assigned to a single voice. As a consequence, Chords can be interpreted homorhythmically.

4.2 MIDI Representation

It is useful to draw a distinction between representation and interpretation. AOMR is rather abstract and focuses on representation, deferring issues of interpretation to a particular AOMR implementation.

In an AOMR implementation, interpretation is the job of those components that transform or "render" an AOMR representation into a music representation that can be visually or audibly perceived. The default implementation of AspectMusic/J contains components that can render CMU and MusicSpace structures as MIDI files. Consequently, it is these components that provide the default interpretation.

These renderers attempt to form MIDI events from the triple of pitch, dynamic level and rhythm. `Pitch`, `DynamicLevel` and `Rhythm` primitives representing the parameters must exist respectively in the classifiers named by the constants `ComposedMusicUnit.PITCH_CLASSIFIER`, `ComposedMusicUnit.DYNAMIC_CLASSIFIER` and `ComposedMusicUnit.RHYTHM_CLASSIFIER`.

4.3 The Logic Gateway

As we have described, AspectMusic/J represents musical events described by JoinPoints as complex object graphs. One disadvantage of this approach is that the logic required to determine whether or not a joinpoint satisfies a given pointcut condition is both procedural and arbitrarily complex.

Asymmetric AOP systems, such as AspectJ, support the notion of "declarative pointcuts". In other words, the pointcut expression states only those conditions that must hold in order for the pointcut to be satisfied.

In order to support declarative pointcuts, AspectMusic utilises the services of the Smalltalk Open Unification Language (SOUL) [7]. AspectMusic/J, in contast, does not prescribe a logic language implementation. Rather, the interface between AspectMusic/J and any given logic language implementation is prescribed by two interfaces (`ILogicRepository` and `ILogicReifier`) and one class (`LogicSolution`). These are defined in the package `aspectmusicj.LogicGateway`.

The two interfaces, `ILogicRepository` and `ILogicReifier`, respectively describe an abstraction of a logic system, and a translator that can convert from AspectMusic/J's object representation to the representation required by that logic system. For convenience, the `LogicGateway` package provides `PrologReifier`, an implementation of `ILogicReifier` for logic systems that use PROLOG's standard syntax.

The class, `LogicSolution`, describes the form that is used by an `ILogicRepository` implementation to convey results from the logic system to AspectMusic/J.

Thus, a logic system may be integrated into AspectMusic/J by providing an implementation of `ILogicRepository`, and, if necessary, `ILogicReifier`. The Java logic system JLog [8] has been found to work well with AspectMusic/J. A Logic Gateway implementation for JLog is provided in the package `aspectmusicj.JLogGateway`.

4.4 Logic Representation of Joinpoint Context

When a MusicSpace is evaluated, the compiler will formulate, at each tick, a Joinpoint context containing an representation of the events whose onset coincided with that tick. In order to use logic-based Aspects it is necessary to understand how a JoinpointContext is expressed using the logic representation.

The JoinpointContext consists of three main parts; metrical location, event description, and composition history.

4.4.1 Metrical Location

The metrical location of a joinpoint context is described by a cuepoint/2 predicate.

```
cuepoint(partName, location(bar,beat,tick))
```

Each cuepoint associates the name the MusicSpace part to which it relates with a location/3 predicate that describes the metrical location (bar, beat, tick) of the joinpoint context with respect to that part.

4.4.2 Event Description

The detailed description of the events whose onset coincides with the joinpoint is given by event/4 predicates.

```
event(partName,classifierName,voiceNumber,value(V))
```

The representation of the event parameter (*V*) depends upon the type of event being represented. Examples of the default Rhythm and Pitch event types are shown below.

```
rhythm(relativeOnset(0),duration(240),nextOnset(0))

pitch(midiPitch(41),pitchClassAndOctave("F3"))
```

The `aspectmusicj.LogicGateway.PrologRefier` implementation uses the following rules to generate event descriptions.

> 1. The event name is the simple name (ie without package name) of the event class, in lower case.
>
> 2. The parameters associated with the event are the elements of the set of getter method names and values for each `@AspectMusicProperty`-annotated getter method. The getter method name is manipulated to drop the initial "get" and convert the first character of the resulting name to lower case.

For example, consider the `aspectmusicj.MusicPrimitives.Pitch` class. By rule 1, the event name for instances of this class is simply "pitch".

The `Pitch` class contains two getter methods annotated as `@AspectMusicProperty`; `getMidiPitch()` and `getPitchClassAndOctave()`. The annotation of latter requires that its values be quoted. Consequently, by rule 2, the logic representation contains two predicates midiPitch/1 and pitchClassAndOctave/1.

4.4.3 Composition History

The composition history associated with each event is represented by a set of history/5 predicates.

```
history(partName,classifierName,voiceNumber,historyItemIndex,H)
```

The partName, classifierName and voiceNumber arguments identify the event to which the history refers. The `historyItemIndex` value is used to establish the sequence of history events for a particular event.

The representation of the history event (H) depends upon the kind of history event.

Repository Location events are represented repositoryLocation/1 predicates, whose single argument is list of param/2 predicates, each representing a key/value pair, describing the location of the event within the repository.

For example, an event which occupied index 1 of voice 1 in the rhythm classifier of the CMU "Q1", located in dimension "Rhythm", concern "Motives" would be represented as follows. Note that the term "dataSet" is used rather than "voice". This is because the repository location event is generated by MDDR rather than AspectMusic/J.

```
repositoryLocation([
      param(identity,"0"),
      param(unit,"Q1"),
      param(dimension,"Rhythm"),
      param(concern,"Motives"),
      param(classifier,"rhythm"),
      param(dataSet,"1")])
```

Transformation events are represented as transform/1 predicates, whose single parameter is a list of param/2 predicates, each representing a key/value pair describing the parameters passed to the transformation. In addition, the value of the key "class" contains the fully qualified name of the transformation class.

The following example shows a transformation event recording that an instance of the TransposeTransform class, with the parameters delta=5 and unitType="pitch", was applied to the event.

```
transform([
  param(unitType,"pitch"),
  param(delta,"5"),
  param(class,
    "aspectmusicj.HyperMusic.StandardTransformer.TransposeTransform")
])
```

5 The MIDI Toolkit

The MIDI toolkit implemented by AspectMusic was a from-scratch implementation of all the MIDI support required by the AspectMusic system. As a consequence this package was quite complex. In contrast, because MIDI support is available for Java (javax.sound.midi), the MIDI toolkit included with AspectMusic/J is somewhat simpler.

The toolkit consists of two key areas of functionality. Firstly support for importing MIDI data into Composed Music Units. Secondly, support for transforming Composed Music Unit and MusicSpace structures into Java MIDI data structures that can be used by external tools.

5.1 Rendering CMUs

CMUs can be rendered as MIDI sequences using an instance of the `aspectmusicj.MidiToolkit.rendering.CMUMidiRenderer` class. This class extracts the pitch and rhythm classifiers from a given CMU and transforms them into a MIDI sequence. Note that this class will provide default pitch or rhythm elements if required.

5.2 Rendering MusicSpaces

MusicSpace instances can be rendered using an instance of the `aspectmusicj.MidiToolkit.rendering.MusicSpaceMidiRendererEx` class. This class renders data from pitch, rhythm, dynamic and harmony classifiers, expecting, respectively `Pitch`, `Rhythm`, `DynamicLevel` and `Chord` primitive instances.

6 Extending AspectMusic/J

AspectMusic/J has been implemented in a way that facilitates extension and modification.

6.1 Writing Transformations

Transformations must be derived from AbstractCDUTransformation. This abstract class contains a number of utility methods used by both the Transformation framework and transformation implementations.

AbstractCDUTransformation declares the abstract method doTransform(). This is the method that will be invoked by AspectMusic/J when a transformation is to be evaluated.

```
   public abstract ComposedDataUnit
doTransform(ComposedDataUnit target)
throws TransformationException;
```

Concrete transformation implementations must therefore implement this method.

The parameter "target" is the ComposedDataUnit (ie CMU) that is to be transformed. The return value is expected to be the transformed CMU. Note that all necessary copy operations are performed by AspectMusic/J. Consequently, the CMU that is passed to doTransform() is a private copy that may be modified in-place and returned, without side-effects.

At the point that AspectMusic/J calls doTransform(), any parameters that have been passed to the transformation will be available in the ParameterSet object returned by getParameters().

6.1.1 Static vs Dynamic Typing

In many cases, transformations will be implemented by calling methods on the objects that represent musical information, contained by the DataItems within CMUs.

Some operations may be common across a number of different objects. For example, a "scale" method might be invoked on objects representing pitch, dynamic level or rhythm. In a statically typed system, such as Java, the implementation of a transformation that can operate on multiple object types is likely to require some mixture of interface implementation, use of `instanceof`, and casting.

AspectMusic/J also supports a dynamic method invocation scheme for transformations. This scheme removes the need for transformation implementations and their target objects to use static-typing contrivances such as those described above. All that is required is for the method, taking a single ParameterSet parameter, to be defined on the target object's class. In order to use this scheme, the transformation implementation simply calls the `invokeTransformMethod` method, inherited from `AbstractCDUTransformation`.

For example, the call

```
invokeTransformMethod(obj,"scale")
```

calls the method `Object scale(ParameterSet)` on the object `obj`, passing a ParameterSet containing any parameters that have been passed to the transformation.

6.1.2 Composition History

Each time an object is transformed, its composition history should be updated to include details of the transformation. The `AbstractCDUTransformation` method `getCompositionHistoryItem()` can be used to obtain a suitable composition history item for any `AbstractCDUTransformation`-derived implementation.

6.2 Custom Primitives

It is possible to add new primitives to AspectMusic/J. There is no base class for primitives, however there are a few requirements that must be met in order for custom primitives to operate correctly within AspectMusic/J.

6.2.1 Integration with Repository Persistence

As has been explained, the Repository persistence mechanism uses Java's XMLEncoder facility to render the object graph represented by a Repository as an XML file. In order to be persisted in this way, primitive classes should follow the JavaBean patterns for property getters and setters, and have a parameter-less constructor.

If this is not possible, then a custom `PersistenceDelegate` must be provided. Customer `PersistenceDelegate`s can be registered with the `HyperMusicRepositoryPersistenceHelper` by calling its `setPersistenceDelegate()` method.

6.2.2 Integration with HyperMusic

As part of the composition mechanism, HyperMusic needs to be able to produce copies of CMUs. HyperMusic therefore requires that any primitive object contained by a CMU can be requested to create a copy of itself. This is achieved by requiring that primitives implement the `DeepCopyable` interface method `Object deepCopy()`.

In practice, if an object is immutable, it may simply return itself. The default primitives `Pitch`, `Rhythm` and `DynamicLevel`, which are immutable, implement `deepCopy()` in this way. The `deepCopy()` implementation of the (mutable) `Chord` primitive, in contrast, constructs and populates a new `Chord` instance.

6.2.3 Integration with the Logic Gateway

The supplied `PrologReifier` implementation transforms primitive instances into a Prolog representation as follows:

```
className(propertyName(propertyValue), propertyName(propertyValue),…)
```

Each JavaBean property that is to form part of the representation must be annotated as `@AspectMusicProperty`. This annotation enables the user to determine whether or not the value should be quoted in the Prolog representation. By default, values are quoted.

7 References

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

[2] P. Hill, "Aspect-Oriented Music Representation," in *Centre for Research in Computing*, vol. Ph.D. Milton Keynes: The Open University, 2007.

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

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

[5] M. Balaban, "Music Structures: Interleaving the Temporal and Hierarchical Aspects in Music," in *Understanding Music with AI*: MIT Press, 1992.

[6] P. Hill, S. Holland, and R. Laney, "Symmetric Composition of Musical Concerns," in *Proceedings of the 5th International Conference on*

Aspect-Oriented Programming (AOSD06). Bonn, Germany: ACM, 2006, pp. 226-236.

[7] W. D. Meuter, J. Brichau, and K. Mens, "SOUL Manual (draft)," Vrije Universiteit Brussel 2006.

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