Patrick Hill, Simon Holland, and Robin Laney
The Open University
Centre for Research in Computing
Walton Hall
Milton Keynes MK7 6AA UK
http://www.computing.open.ac.uk/Themes/HCI
PatrickHill@bcs.org.uk
{s.holland, r.c.laney}@open.ac.uk

An Introduction to Aspect-Oriented Music Representation

Computer-music composition systems serve diverse purposes. Some draw on contrasting computing paradigms, including procedural, object-oriented, data-flow, functional, and logic programming. All such systems, provided they are sufficiently developed, are Turing-complete, but different paradigms can help facilitate different kinds approaches to music computing and new kinds of music manipulation.

In general-purpose programming, there is a recent development known as aspect-oriented programming (AOP) that has been claimed, along with related techniques,to offer new kinds of flexibility and ease of variation. The system presented here, AspectMusic, is the first substantial work we are aware of to adapt ideas from aspect-oriented programming and related techniques and explore how they might be applied to facilitate new approaches to musical exploration and manipulation. AspectMusic is an implemented framework that applies ideas at the root of AOP to the manipulation of musical materials and structures. To introduce AspectMusic,and more generally, aspect-oriented music representation (AOMR), we begin by reflecting on some various well-known features of current computer-music systems.

Interactive scoring systems effectively provide musical "word-processor"-style environments, enabling musical detail to be entered and edited on a note-by-note basis. Other approaches, such as JMusic and Common Music, attempt to provide a generalized music programming environment, albeit with a prescribed musical ontology, embedded within general-purpose programming languages. Computer music environments such as Pope's MODE system (Pope 1991) have evolved out of the requirement to support particular compositional needs, and therefore although multiple approaches are supported, ontological generality is not a prime concern.

Computer Music Journal, 31:4, pp. 47–58, Winter 2007
© 2007 Massachusetts Institute of Technology.

In these and other ways, most music representations allow music to be represented and manipulated only within a prescribed range of preconceived musical ontologies. In contrast, AOMR has the distinctive aim of supporting whatever ontologies may be preferred by particular users. This is largely achieved by abstracting music constructs in terms of discrete areas of interest, or *concerns*, that may be composed together. In this way, AOMR attempts to support any perspective as a first-class entity; its operations, however, may crosscut or be scattered across existing organizational hierarchies.

In this article, we have focused on familiar and relatively simple examples using Western tonal music to explain AOMR as clearly as possible. For clarity, we proceed from the traditional dimensions of tonal music, and then we show how AOMR systematically supports various operations and conceptions that crosscut these dimensions.

AOMR appears to be particularly relevant for those musical idioms in which musical composition may be characterized in terms of a typically limited set of musical "raw materials" that are combined and reused in various ways, addressing different musical areas of interest within a particular piece of music. Well-known examples of such materials in Western tonal music include pitch sequences, rhythmic figures, and harmonic progressions that may appear in exact and transformed forms throughout a musical work. Similar transformational processes can occur at abstract levels in other idioms, with materials such as sequences of parameters that form the inputs to generative processes. Equally, AOMR is applicable to musical works and fragments based on structural prototypes that are formed through a finite set of combinatorial operations. AOMR appears applicable wherever explicit support for maintaining coherence across representational or functional shifts is useful to a composer.

In traditional areas such as those idioms we focus

Downloaded from http://direct.mit.edu/comj/article-pdf/31/4/47/1854942/comj.2007.31.4.47.pdf by guest on 17 September 2021

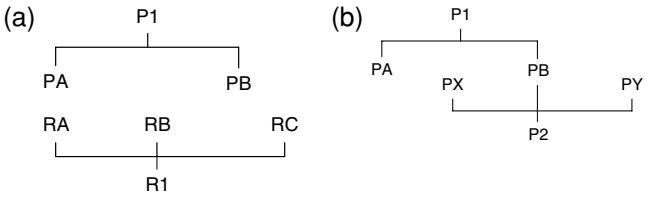

Figure 1. (a) Tangled hierarchical relationship of pitch and rhythm; (b) polyarchic pitch relationship.

on in this article, it is generally argued that the reuse of materials such as those just mentioned is the principal method by which a composer achieves musical coherence and stability across different musical dimensions (Schoenberg 1967; Sloboda 1985; Cook 1987; Belkin 1999). Additionally, of course, composers may reuse materials inter-opus. In relevant styles, the general process of musical composition therefore typically requires that the composer merge together selected musical fragments to form new constructs. As a consequence, musical materials cannot generally be localized to any particular musical construct.

To take a simple example, non-problematic using conventional systems, a single melodic figure might be merged with a number of rhythmic variants, forming different phrases. Musical materials therefore tend to be explicitly restated and transformational processes re-applied at each occurrence. Hierarchical structures are prominent in music, as indicated by the analytical work of theorists such as Schenker as well as Lerdahl and Jackendoff (1983). However, because the hierarchies formed in different dimensions do not necessarily align, musical compositions tend to contain *tangled hierarchies* and *polyarchies.* For example, consider a simple pitch sequence P1 consisting of two parts PA and PB, and a simple rhythmic sequence R1 consisting of three parts RA, RB, and RC. A single melodic figure constructed from these sequences results in a tangled hierarchic relationship in which there is not necessarily any correspondence between any of the parts of P1 and those of R1. This is shown in Figure 1a. Polyarchic relationships occur when a node is shared between hierarchies. Consider, for example, another pitch sequence P2 that also uses PB, shown in Figure 1b. Tangled polyarchies occur when elements from polyarchies in different dimensions are combined.

In addition to deriving core musical materials, in composing and arranging a musical work composers often make associations between musical context and events occurring in orthogonal dimensions. For example, another simple and non-problematic example is the modification of dynamic level at a particular point within a piece of music, either based upon metrical location, or musical context, such as the occurrence of a particular chord type. More sophisticated examples include the derivation of parts from musical events occurring in one or more other parts, such as accentuating particular notes through doubling. A key point is that the potential complexity of hierarchical tangling that may arise from such derivations is, in principle, unlimited. Also, even simple hierarchical tanglings can lead to unlimited complexity when multiplied or iterated. An equally important related point is that, in a finished score, compositional intent is often lost, and again these derivation processes must be restated and reapplied wherever they recur. Moreover, if musical fragments are to be specially treated, based upon features relating to their derivation, then this too must be tracked by the composer because it may not be readily apparent from the resultant material itself (Crochemore, Iliopoulos, and Pinzon 2001).

Furthermore, whereas music is naturally experienced in a "left-to-right" fashion, it is not necessarily composed in this way. Although it is true that some composers speak of a spontaneous vision of a work that "simply" requires transcription (Sloboda 1985), others (e.g., Spiegel 1988) describe detailed, iterative, evolutionary processes, in which the composition develops from possibly incomplete sketches in different musical dimensions and at different levels of abstraction.

The representation of tangled, multi-dimensional, polyarchic relationships is difficult and generally not explicitly represented in computer-music systems. However, recent developments in computer software, collectively termed aspect-oriented programming (AOP), seek to address similar types of problem that exist in software. In particular, AOP approaches try to help manage the separation, encapsulation, and subsequent merging together, or *weaving,* of the implementations of separately

Downloaded from http://direct.mit.edu/comj/article-pdf/31/4/47/1854942/comj.2007.31.4.47.pdf by guest on 17 September 2021

specified areas of interest or *concern.* Modern object-oriented approaches to software design and implementation (e.g., Meyer 1997; Rumbaugh et al. 1991) represent a natural evolution of the module paradigm suggested by Parnas (1972), enabling application-domain and design concerns to be represented as classes that encapsulate both data and the operations that may be performed upon that data.

Even so, certain areas of interest remain difficult or impossible to encapsulate as classes (Hürsch and Lopes 1995). From one perspective, the fields and methods defined within a class might be further grouped in terms of the concerns that they address and the implementation of a given concern might involve a number of methods and fields that extend across multiple, possibly unrelated, classes. Moreover, multiple concern implementations may incorporate common code fragments that themselves have no natural class association. Typical object-oriented approaches therefore impose a decomposition scheme that is not sufficiently general to enable separation of concerns. This has been dubbed "the tyranny of the dominant decomposition" (Tarr and Ossher 2000). At another level of abstraction, certain programming concerns relate to particular events that occur within the execution of a program. For example, a "tracing concern" that produces diagnostics that trace entry into selected methods typically requires the tracing implementation to be restated in every method of interest.

These complementary viewpoints identify that certain concerns tend to be *scattered* across multiple, possibly unrelated, classes and intertwined or *tangled* with other concern implementations. Because such concerns cannot be localized and encapsulated using the prevailing decomposition, they are said to be *crosscutting.* Addressing such problems is precisely the purpose of AOP and cognate techniques.

In this article, we introduce an experimental object-oriented (OO) software system called AspectMusic that adapts ideas from AOP to realize an aspect-oriented music representation. AOMR aims to provide a general approach to the separation, organization, and composition of musical concerns from both viewpoints outlined herein.

Symmetric and Asymmetric Composition of Concerns

Approaches to AOP are diverse. Some, such as Aspectual Components (Lieberherr, Lorenz, and Mezini 1999) apply to specific programming concerns, whereas others, such as AspectJ (Xerox 2002), Hyper/J (Tarr and Ossher 2000), and Caesar (Mezini and Ostermann 2003) address more general separation of concerns. General-purpose approaches broadly adopt one of two paradigms: symmetric or asymmetric. The following paragraphs briefly outline these approaches. In each case, the reader is directed to the references for further detail. Examples of these two approaches as applied to music are presented in later sections of this article.

Asymmetric approaches, such as AspectJ (Xerox 2002) and AspectS (Hirschfeld 2002), consider the augmentation of a base decomposition—typically a standard OO-decomposed program—with crosscutting concern implementations. In AspectJ (the best-known example of this approach), the base decomposition is a normal Java program. Crosscutting concern implementations, generically termed *advice,* are associated with well-defined points, termed *joinpoints,* in the static or dynamic structure of the base decomposition. Typically, joinpoints are at method calls and variable assignments. Particular joinpoints of interest are specified as *pointcut* expressions, and advice may typically be run before, after, or instead of (*around*) the joinpoint. In this way, asymmetric approaches in general enable standard programs to be non-invasively augmented with separately encapsulated crosscutting concern implementations.

In contrast, symmetric approaches, such as multidimensional separation of concerns (MDSoC; Ossher and Tarr 1999) implemented by Hyper/J (Tarr and Ossher 2000), consider the construction of software systems from separately specified components (*units*) that each address just one concern. In MDSoC, these components are organized into a multidimensional structure called a *hyperspace,* in which each named unit is associated with a dimension name, and the name of a concern within that dimension to which the unit relates. Broadly speak-

Downloaded from http://direct.mit.edu/comj/article-pdf/31/4/47/1854942/comj.2007.31.4.47.pdf by guest on 17 September 2021

ing, using an MDSoC approach, software systems are composed by specifying those dimensions and concerns that the resultant system should contain. As noted in Harrison, Ossher, and Tarr (2002), no single composition method is suitable for all requirements. Therefore, symmetric and asymmetric approaches should be regarded as complementary.

AspectMusic Overview

AspectMusic is an object-oriented, experimental implementation of an AOMR written in VisualWorks Smalltalk (Goldberg and Robson 1989). The system consists of two interrelated components, HyperMusic and MusicSpace, that respectively implement symmetric and asymmetric composition approaches.

HyperMusic provides a framework for abstracting, organizing, and representing musical ideas. Using HyperMusic, the user defines musical materials and transformational processes that are then organized according to user-defined criteria into a hyperspace. Declarative specifications are used to compose new material from the components in the hyperspace. These new components may themselves be added to the hyperspace, and thus the hyperspace becomes an evolving repository of compositional ideas.

MusicSpace allows components from HyperMusic to be arranged in time, in a manner similar to a MIDI sequencer. However, MusicSpace enables music to be dynamically modified based upon context using an approach that is similar to asymmetric AOP. In particular, modification-strategy implementations can be modularized and associated with particular dynamic musical conditions in terms of pointcut-like events that must be satisfied for the strategy to be applied.

To link the two approaches, the construction of materials using HyperMusic is audited through a "Composition History" that is associated with each symmetrically composed element. This history is available to MusicSpace. Thus, for any musical event in MusicSpace, it is possible to determine not only its current state, but also how it has been derived. In the remainder of this article, we describe the symmetric and asymmetric components of AspectMusic in greater detail.

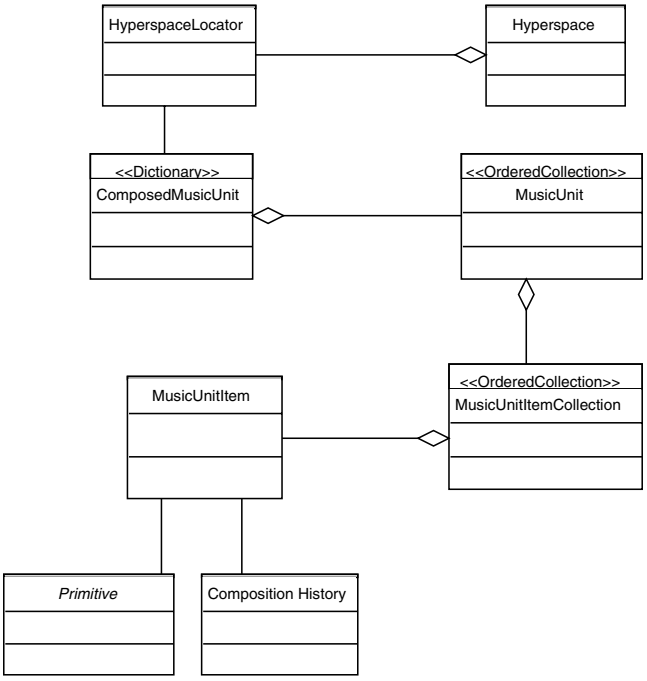

Figure 2. HyperMusic class structure.

Symmetric Composition of Musical Concerns Using AspectMusic

The symmetric-composition component (HyperMusic) of AspectMusic provides an extensible framework that supports the definition, organization, and combination of musical fragments (Hill, Holland, and Laney 2006). HyperMusic is heavily influenced by the notion of hyperspaces (Ossher and Tarr 1999) and the Hyper/J (Tarr and Ossher 2000) implementation of hyperspaces for the Java programming language. The principal class structure of HyperMusic is shown in Figure 2.

Structuring in HyperMusic

Fundamental to HyperMusic is its ability to support the separation and combination of musical ideas from discrete musical dimensions. In this context, a

Downloaded from http://direct.mit.edu/comj/article-pdf/31/4/47/1854942/comj.2007.31.4.47.pdf by guest on 17 September 2021

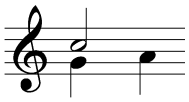

Figure 3. A simple music fragment.

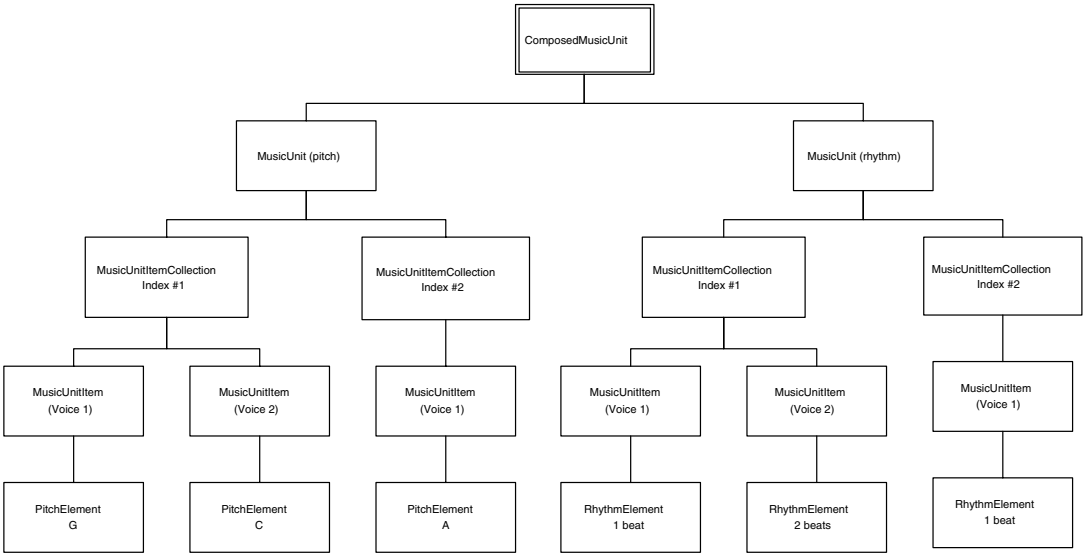

Figure 4. Schematic of a CMU representing the fragment shown in Figure 3.

musical idea is generalized as an indexed collection of events relating to the same musical dimension, where each event may contain multiple elements organized into discrete voices. For example, a monophonic rhythmic idea might be expressed as a sequence of events, where each event contains a single object representing relative onset time and duration. A sequence of chords, on the other hand, might be expressed as a sequence of events in which each event contains pitch-value representations distributed across multiple voices.

In HyperMusic, music units (MUs) represent discrete musical ideas in a single musical dimension, organized as an ordered collection. Each element of the ordered collection within an MU contains an ordered collection (a `MusicUnitItemCollection`) of `MusicUnitItem` objects that wrap user-defined objects representing musical information. The `MusicUnitItemCollection` indexes its `MusicUnitItems` with values that correspond to discrete voices. Conceptually, the wrapped objects each pertain to the same musical *type,* such as pitch, rhythm, or dynamic information. However, HyperMusic does not prescribe the types that can be represented; neither does HyperMusic mandate that all wrapped objects within an MU are of the same class. For example, a pitch MU might contain `MusicUnitItems` that wrap a mixture of object types that represent pitch as, say, MIDI pitch values (0–127), symbolic values such as "C#4," or frequency values.

Complete musical ideas are typically formed from multiple musical dimensions. A melody, for example, can be viewed as being formed from pitch and rhythm. In HyperMusic, MUs of different types are aggregated into composed music units (CMUs), represented by the class `ComposedMusicUnit`, in which each component MU is identified by the name of the type it represents. To illustrate the structure of CMUs, a simple musical fragment and a schematic representation of the HyperMusic objects involved in a CMU representation of the fragment are shown, respectively, in Figures 3 and 4.

Combining CMUs

CMUs are designed to be combined in various ways to form new CMUs that represent new musical material. We adopt the AOP term *weaving* to describe these combination processes. HyperMusic supports the musically fundamental notions of sequential and parallel weaving. However, it is easy to extend the system to support other kinds of weaving corresponding to any method through which two CMUs can be combined, for example weaving appoggiatura or acciaccatura (Honing 1993)

Downloaded from http://direct.mit.edu/comj/article-pdf/31/4/47/1854942/comj.2007.31.4.47.pdf by guest on 17 September 2021

configurations. It is important to note that CMUs are not constrained to be "complete." Although clearly for final realization, CMUs must contain all required musical elements in all required dimensions, at any intermediate stage a CMU can be incomplete in terms of the types that it contains (vertically) and the number of elements in each type (horizontally). Because there is no implied correspondence among arbitrary groupings within each type, CMUs are able to support tangled interdimensional relationships.

Sequential Weaving

When two CMUs are woven in *sequence,* notated with the binary operator +, those MUs with identical type names are concatenated. If an MU type appears in only one of the woven CMUs, then it is concatenated with an empty MU. Thus, the set of MU types contained in the resultant CMU is the union of the set of MU types of both the combined CMUs. Usefully, this means that if, say, a CMU containing only a pitch dimension is sequentially woven with a CMU containing only rhythm, then the resultant CMU contains the pitch sequence in *parallel* with the rhythm sequence.

Parallel Weaving

When two CMUs are woven in parallel, notated with the binary operator `|`, the `MusicUnitItemCollections` that exist at the same index in MUs of the same type name are merged together, allowing, for example, the formation of chords. Like sequential weaving, the resultant CMU contains the union of the set of MU types of both combined CMUs.

Transformations

In the foregoing sections, we have described the representation of musical data within the CMU model. However, music composition often involves the algorithmic transformation or generation of musical material. The CMU model supports these requirements by enabling transformational or generative code, generically termed *transformations,* to be placed in the special MU type `#Transform`. Like other types, the `#Transform` type is represented by a `MusicUnit` structure, enabling ordering to be represented. A common use might be to construct CMUs that contain just one or more transformations. However, as we will demonstrate later, the open-ended nature of the CMU structure means that the approach also naturally supports CMUs that contain both transformational code and other musical information.

Combining and Executing Transformations

Although, like other types, the `#Transform` type is an MU, the semantics of non-sequential weaving are not defined for it. Consequently, `#Transform` is *always* woven sequentially. However, simply weaving transformations into a CMU does not cause the transformations to be applied. Rather, transformations are executed by *evaluating* the CMU. This approach enables sequences of transformations to be constructed and allows the user to specify when these transformations should be applied. Evaluation, notated with the unary operator `@`, causes all transformations in a CMU's `#Transform` MU to be executed in index order and for a resultant CMU to be produced that contains no `#Transform` type. A CMU without a `#Transform` type is said to be *final,* and the result of evaluating a final CMU is the CMU itself, unchanged.

Organizing and Combining CMUs

CMUs represent raw musical data and transformational processes. However, CMUs are of limited value in themselves. Rather, we want to be able to organize CMUs according to our own ontology, and by combination, to construct new CMUs by reference to this organization. The organizational structure of HyperMusic, like that of MDSoC, is the *hyperspace.*

Dimensions and Concerns

In the hyperspace model presented in Ossher and Tarr (1999), units are organized according to the di-

Downloaded from http://direct.mit.edu/comj/article-pdf/31/4/47/1854942/comj.2007.31.4.47.pdf by guest on 17 September 2021

mension and concern to which they relate. In this context, dimension and concern are arbitrary textual names whose purpose is to represent an organizational structure that transcends the explicit structure of the software itself. In object-oriented software, for example, while a *class* may represent a prototypical object that exists within some universe of discourse, the fields contained within the class—and the operations that may be performed on instances of that class—may conceptually pertain to different functions or concerns of the system. Moreover, the operations that pertain to a given concern may exist in multiple classes. Thus, because the dominant, class-based, decomposition of object-orientation does not support such a partitioning, it is not easily possible in object-oriented programming to identify only those parts of the class graph that relate to a particular concern.

There are many established musical representations, each with their own musical ontology, and consequently there is no clear analog of a "dominant decomposition" in music. Nonetheless, music can be considered in terms of independent perceptual dimensions (Loy and Abbott 1985): pitch, rhythm, dynamics, and timbre. We may also consider music in terms of aggregates of a single dimension, such as melodic themes, cells, tone rows, harmonic progressions, rhythmic motifs, or—more abstractly—arbitrary parameters and their evolution through generative, transformational, and combinatorial processes throughout a given musical work. We argue, therefore, that for many purposes it is useful to consider each newly generated musical structure as serving some identifiable compositional purpose that can be attributed to some dimension and concern within the work. For example, different melodic fragments might be used in an antecedent/consequent relationship, themes might be associated with extramusical events in similar ways to Wagner's use of the *leitmotiv.*

Hyperspace and Hyperslices

In HyperMusic, a hyperspace is a data structure that maps between coordinates (consisting of the triplet of *dimension* name, *concern* name, and *unit* name) and CMUs. Thus, the hyperspace, represented by the `Hyperspace` class, is an organized repository of musical and transformational fragments represented as CMUs.

A *hyperslice* is an abstract slice through a hyperspace, defined by a partial specification of the coordinates of CMUs of interest. For example, we might be interested in any CMU in the "Themes" dimension, or any CMU in a concern whose name begins with "Ostinato."

Combining CMUs

The purpose of organizing CMUs into a hyperspace is to facilitate their subsequent composition into new CMUs. The specification of such a composition, represented by an object of the `Hypermodule-Specification` class, contains three key attributes: one or more *hyperslice specifications,* a *composition expression,* and a *composition relationship specification.*

Composition Expressions

A *composition expression* specifies the hyperspace coordinates of those CMUs that are to be woven and the weaving operations that are to be performed on these CMUs to generate a new CMU. Typical weaving operations, as described previously, are sequence (+) and parallel (||). Additionally, the composition expression may cause the evaluation of CMUs, using the unary operator `@`. Within the composition expression, CMU coordinates are specified in the form `dimension;concern;unit`, where `dimension`, `concern`, and `unit` are regular expressions.

For example, consider the two CMUs `A` and `B`, shown in Figure 5, and the various weavings of these CMUs, shown in Figures 6–8. For the purposes of this example, assume that `A` and `B` exist in the `C` concern of dimension `D`. Assume also that `A` and `B` each contain some musical information in the `#pitch` dimension, and that `A` additionally contains a transformation `T` that transposes pitch up by one semitone.

As shown in Figure 6, the composition expression `D;C;A + D;C;B` causes `A` and `B` to be sequentially

Downloaded from http://direct.mit.edu/comj/article-pdf/31/4/47/1854942/comj.2007.31.4.47.pdf by guest on 17 September 2021

Figure 5. CMUs `A` *and* `B`.

Figure 6. CMU resulting from the composition `D;C;A + D;C;B`.

Figure 7. CMU resulting from the composition `@D;C;A`.

Figure 8. CMU resulting from the composition `@(D;C;A) + D;C;B`.

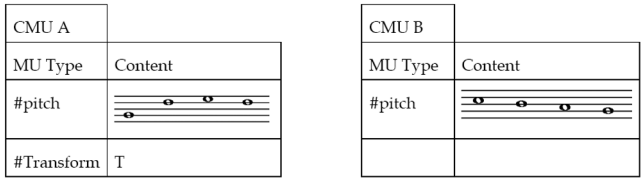

Figure 5

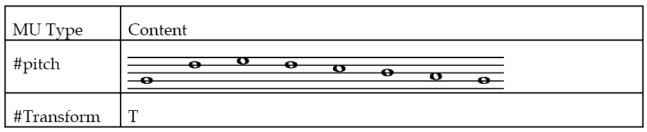

Figure 6

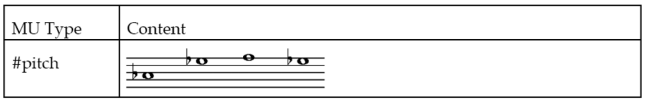

Figure 7

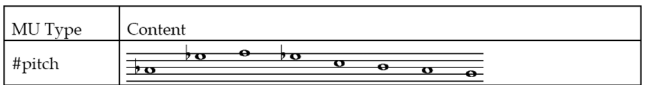

Figure 8

woven, resulting in a CMU whose content is the pitch component of `A` followed by the pitch component of `B`, and that also contains the transformation `T`.

The expression `@D;C;A` results in a CMU that contains the pitch component of `A` that has been transposed up by one semitone, owing to the evaluation (`@`) of `T`. The evaluation also causes `T` to be removed from the resultant CMU. This is shown in Figure 7. The expression `@(D;C;A) + D;C;B` results in a CMU containing the pitches of `A` transposed up by one semitone, followed by the (untransposed) pitches of `B`, as shown in Figure 8.

Although specifying CMUs in terms of absolute hyperspace coordinates is clearly useful, greater expressive power is achieved from partial specifications using regular expressions. By default, CMUs are woven in sequence. For example, the composition expression `Intro;BassGuitar;Fragment.*` represents the sequential weaving of all CMUs whose name begins with `Fragment` and that exist in the `BassGuitar` concern of the `Intro` dimension, subject to the hyperslice constraints subsequently described.

Hyperslice Specifications

To select CMUs for weaving, the CMU coordinates specified by a composition expression must be matched against CMUs that exist in the hyperspace. *Hyperslice specifications* within a hypermodule specification cause the search space to be refined by restricting matched names to those existing in the specified hyperslices rather than the entire hyperspace. Hyperslice specifications can be expressed as regular expressions. Figure 9 shows the content of several example hyperslices of the following Hyperspace

```
Themes;Theme1;Opening1
Themes;Theme1;Closing1
Rhythms;Theme1;Original
Rhythms;Theme2;Original
Rhythms;Theme1;1stVariation
```

Composition Relationships

Because composition expressions may specify CMU coordinates as regular expressions, it is possible for multiple CMUs to match the specification. A *composition relationship* specifies how this situation is managed. HyperMusic contains two predefined composition relationships, though it is possible to add new relationships as required. In the event of multiple matches for a CMU, the `overrideByName` relationship causes the last match found to be used. Conversely, the `mergeByName` relationship causes all matching CMUs to be woven in sequence. The results of the matching process are always returned in ascending ASCII order. Thus, in a hyperspace containing CMUs given by

```
Dim.Concern.Unit1
Dim.Concern.Unit2
Dim.Concern.Unit3
```

Downloaded from http://direct.mit.edu/comj/article-pdf/31/4/47/1854942/comj.2007.31.4.47.pdf by guest on 17 September 2021

Figure 9. Example hyperslice specifications and their content.

Hyperslice Specification	Hyperslice Content
Themes;.*	Themes;Theme1;Opening1 Themes;Theme1;Closing1
.*;Theme1	Themes;Theme1;Opening1 Themes;Theme1;Closing1 Rhythms;Theme1;Original Rhythms;Theme1;1stVariation
Rhythms;Theme1	Rhythms;Theme1;Original Rhythms;Theme1;1stVariation

Themes;Theme1;Opening1

the regular expression `Dim;Concern;Unit.*` would be matched by all three CMUs. If the `overrideByName` relationship were used, then `Dim.Concern.Unit3` would be returned. If the `mergeByName` relationship were used, then a CMU formed by the sequential composition of `Dim.Concern.Unit1 + Dim.Concern.Unit2 + Dim.Concern.Unit3` would be returned.

Composition History

An important feature of music is that musical ideas evolve both inter- and intra-opus. The HyperMusic approach enables evolution to be defined declaratively as hypermodule specifications that define the weaving or transformation of existing CMUs to form new CMUs. As part of the composition of CMUs, each `MusicUnitItem` is annotated with a "composition history" that provides an "audit trail" of the various weavings and transformations that have been performed and have caused the item to be in its present location.

Refactoring

Music composition is obviously a creative and generally iterative process. As such, decisions that are made at some point in the process might be modified at a later point. For example, a CMU might be constructed as a single rhythmic figure `R`. At some later point, it might be decided that `R` can be usefully considered as two components, `R1` and `R2`, and that `R1`, say, can be reused within the composition. We term this kind of adjustment *refactoring.*

In its present form, HyperMusic does not contain any support for automatic refactoring operations. However, if we imagine the composition of CMUs being described as a sequence of weaving operations, then it is a simple matter to adjust this sequence such that `R` is described as being constructed from `R1` and `R2`, thus making `R1` and `R2` available within the hyperspace but leaving all existing references to `R` intact.

Asymmetric Composition of Musical Concerns using AspectMusic

The HyperMusic approach to AOMR seeks to enable musical material, encapsulated as CMUs, to be represented, organized, and woven together to form new material through declarative specifications. Although it is possible to realize CMUs as, for example, MIDI sequences, the range of musical compositions that can be represented in this way is naturally limited.

MusicSpace provides an environment in which the events contained within CMUs can be arranged in time. In this respect, MusicSpace resembles a typical MIDI sequencer. However the distinguishing feature of MusicSpace is its ability to enable music to be modified in relation to context through an approach that mirrors asymmetric aspects in software. The chief advantages of this approach are, first, that modification processes that occur at multiple locations (and are therefore crosscutting) can be modularized, requiring them to be stated only once, and second, aspects are non-invasive—there is no requirement to make any particular a priori arrangements within a CMU for aspects to be applied to the content of that CMU.

The MusicSpace approach enables compositional intent to be explicitly and declaratively expressed. For example, consider Beethoven's *Sonata in C Minor No. 8, Op. 13* ("Pathetique"). At bars 5–7, there is a fragment in which the theme is played *pp* and interspersed with *ff* interludes. We might suggest that Beethoven's structural plan was to have *pp*

Downloaded from http://direct.mit.edu/comj/article-pdf/31/4/47/1854942/comj.2007.31.4.47.pdf by guest on 17 September 2021

sections followed by *ff* sections, irrespective of what musical content these sections ended up containing. Alternatively, Beethoven might have considered that the *ff* markings were to be associated with the musical content of the interludes, irrespective of where they were placed. Perhaps, Beethoven wanted to accentuate those sections that were played in the lower registers, and so *ff* should be applied to any section in which all parts fell below middle C. From an analytical perspective this is pure speculation, with some explanations being more plausible than others. Clearly, whatever the motivation, the resultant score would look the same, and the composer's intent has therefore been lost.

An example from a more contemporary style might be to consider a section of a musical work containing pitch clusters that have been generated using a transformation based upon, for example, three parameters; mode, cluster size, and root. As part of this work, the composer wishes to reinforce particular pitches through doubling on another instrument, depending, say, upon the mode of the cluster in which they appear.

Using MusicSpace, the special treatment of events, such as alteration of dynamic level or replication to a different instrumental part, may be separately and reusably encapsulated. Moreover, the compositional intent that associates such additional behaviors with specific conditions, which may include not only features of specific events, but also features of their derivation, may be declaratively specified.

MusicSpace Structure

A MusicSpace can be considered as a container for one or more named MusicSpaceParts. A MusicSpacePart is analogous to a part on an instrumental score, or a track within a MIDI sequencer, and contains CMUs arranged in time, with the restriction that CMUs cannot overlap within a MusicSpacePart. Figure 10 shows the basic structure of a MusicSpace that has been populated with two MusicSpaceParts: "Melody" and "Bass." The CMUs within a part cannot overlap; CMUs can, however, overlap between parallel parts. Note also that

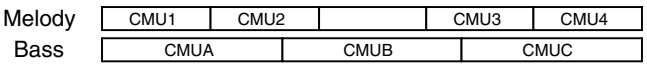

Figure 10. MusicSpace structure.

CMUs are not constrained to be contiguous within a part.

Joinpoints

As in software, some of types of crosscutting concerns, termed *code-level crosscuts* (Bockisch et al. 2004), can be directly mapped to loci in the base decomposition. For example, referring to the logging example outlined earlier, it would be a relatively simple matter to automatically modify a program's source code by inserting logging calls into the methods of interest. This type of crosscut could be easily implemented (with the potential loss of traceability and intent) as a search-and-replace-style function operating on MusicSpacePart objects, for example, modifying duration to achieve articulation effects. However, *dynamic crosscuts* (Bockisch et al. 2004), whose loci can only be determined at run time, present a potentially more useful class of crosscutting concern. In the case of logging, for example, we might be interested in logging one method but only if it has been called by another method. Similarly, we might be interested in making particular notes in a given CMU staccato, but only when they are played at the same time as another CMU. To support dynamic crosscutting, the aspect model used in MusicSpace is *event-based,* conceptually resembling Axon (Aussmann and Haupt 2003) and the Event-Condition-Action (ECA) type "triggers" that are common in database applications. To generate joinpoint events and process aspects, the MusicSpace is compiled against an AspectManager object with which all required aspect objects have been registered.

When a MusicSpace is compiled, "tick" events are generated and propagated to all MusicSpaceParts. These ticks, which by default are generated at a rate of 480 times per beat (giving good resolution of *n*-tuplets), simulate the ticks of a clock source, such as a MIDI clock. However, it should be noted that, in the current implementation, ticks are not

Downloaded from http://direct.mit.edu/comj/article-pdf/31/4/47/1854942/comj.2007.31.4.47.pdf by guest on 17 September 2021

generated in real time, and therefore MusicSpace is not a live-performance environment.

MusicSpace Aspects

MusicSpace aspects are implemented as standard Smalltalk classes that define crosscutting behavior (advice), along with a representation of the conditions under which this behavior is to be executed (pointcuts). An instance of each such class is registered with the AspectManager against which the MusicSpace is to be compiled.

At each clock tick, the contents of all MusicSpaceParts within the MusicSpace are queried to build a *context* that describes all the events that commence at the current tick value. This context includes all of the composition history that is associated with each event. The advice associated with a MusicSpace aspect is invoked if the current context, including temporal location, satisfies the pointcut expressions associated with the aspect. In general, an AspectMusic aspect can modify the content of the current temporal location by manipulating the events contained within the context. The advice may also modify future (or prior) events within the MusicSpace itself. Both *before* and *after* advice types are supported. *Before advice* is executed before the MusicSpace events contained by the current context are rendered into the resultant MusicSpace, enabling the advice to veto or modify the rendering of any component event of the context. *After advice* executes once this process has been completed, and therefore cannot modify the current event.

One of the perceived benefits, particularly of asymmetric AOP, is that pointcuts are defined declaratively. As MusicSpace is an object-oriented Smalltalk system, determining whether a pointcut is satisfied or not often requires procedural code that navigates and queries the object structures of the joinpoint context. These kinds of query may be arbitrarily complex and are, therefore, difficult to generalize in an object-oriented system. This kind of problem is, largely, more easily solved using logic programming languages such as PROLOG. To support declarative pointcut queries, MusicSpace enables joinpoint contexts, including composition history, to be exposed as a set of AspectMusic logic predicates expressed in the PROLOG-like Smalltalk Open Unification Language (SOUL; Gybels 2001). The symbiosis that exists between Smalltalk and SOUL enables expressions in either language to be evaluated from the other.

Rendering

The current implementation of AspectMusic enables MusicSpace instances to be rendered as MIDI sequences by transforming information relating to pitch, rhythm, and dynamics into MIDI events. This provides a convenient method of auditioning and visualizing generated musical compositions using external MIDI tools. However, it should be noted that AOMR and AspectMusic systematically avoid the limitations of any design bias towards particular rendering means such as MIDI.

Conclusions

We have proposed an approach to music representation that is particularly applicable to styles of music in which musical ideas across various musical and non-musical dimensions are merged together to form new musical elements, resulting in tangled, polyarchic relationships. We have described aspect-oriented music representation, which draws from aspect-oriented programming, as a means to explicitly and declaratively support the organization and combination of musical and procedural elements separately from the musical material itself. AOMR supports the symmetric composition of musical materials in which no one musical dimension is dominant, using a musical hyperspace as a dynamic repository of musical ideas. Moreover, AOMR does not limit the kinds of musical information that can be represented. AOMR also features an asymmetric, context-dependent, joinpoint-interception model through which the selective modification of these materials is possible. Because the implementation of our AOMR system is open, the full expressive power of a general-purpose programming language—in this case, Smalltalk—is available within AOMR.

Downloaded from http://direct.mit.edu/comj/article-pdf/31/4/47/1854942/comj.2007.31.4.47.pdf by guest on 17 September 2021

An AspectMusic tutorial is available in the Open University Department of Computing Technical Report TR2006/12 (available online at computing-reports.open.ac.uk/index.php/2006/200612).

References

Aussmann, S., and M. Haupt. 2003. "Axon–Dynamic AOP through Runtime Inspection and Monitoring." Technical Report. Darmstadt, Germany: 2003 ASARTI Workshop.

Belkin, A. 1999. *A Practical Guide to Musical Composition.* Available online at www.musique.umontreal.ca/personnel/Belkin/bk/.

Bockisch, C., M. Haupt, M. Mezini, and K. Ostermann. 2004. "Virtual Machine Support for Dynamic Join Points." *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development.* New York, New York: Association for Computing Machinery, pp. 83–92.

Cook, N. 1987. *A Guide to Musical Analysis.* London: Oxford University Press.

Crochemore, M., C. S. Iliopoulos, and Y. J. Pinzon. 2001. "Computing Evolutionary Chains in Musical Sequences." *The Electronic Journal of Combinatorics* 8(2). Available online at www.combinatorics.org/Volume_8/v8i2toc.html.

Goldberg, A., and D. Robson. 1989. *Smalltalk-80: The Language.* Boston, Massachusetts: Addison-Wesley.

Gybels, K. 2001. "Aspect-Oriented Programming using a Logic Meta-Programming Language to Express Cross-Cutting Through a Dynamic Joinpoint Structure." Licentiate Dissertation, Vrije Universiteit Brussels.

Harrison, W., H. Ossher, and P. Tarr. 2002. "Asymmetrically vs. Symmetrically Organized Paradigms for Software Composition." Technical Report RC22685 (W0212-147). Yorktown Heights, NY: IBM.

Hill, P., S. Holland, and R. Laney. 2006. "Symmetric Composition of Musical Concerns." *Proceedings of the 5th International Conference on Aspect-Oriented Programming (AOSD06).* New York, New York: Association for Computing Machinery, pp. 226–236.

Hirschfeld, R. 2002. "Aspect-Oriented Programming with AspectS." Technical Report. Munich, Germany: DoCoMo Communications Laboratories Europe.

Honing, H. 1993. "Issues in the Representation of Time and Structure in Music." *Contemporary Music Review* 9:221–239.

Hürsch, W., and C. Lopes. 1995. "Separation of Concerns." Technical Report NU-CCS-95-03. Boston, Massachusetts: College of Computer Science, Northeastern University.

Lerdahl, F., and R. Jackendoff. 1983. *A Generative Theory of Tonal Music.* Cambridge, Massachusetts: MIT Press.

Lieberherr, K., D.-H. Lorenz, and M. Mezini. 1999. "Programming with Aspectual Components." Technical Report NU-CCS-99-01. Boston, Massachusetts: College of Computer Science, Northeastern University.

Loy, G., and C. Abbott. 1985. "Programming Languages for Computer Music Synthesis, Performance, and Composition." *ACM Computing Surveys* 17(2):235–265.

Meyer, B. 1997. *Object-Oriented Software Construction, 2nd Edition.* Upper Saddle River, New Jersey:Prentice-Hall.

Mezini, M., and K. Ostermann. 2003. "Conquering Aspects with Caesar." *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development.* New York: Association for Computing Machinery, pp. 90–99.

Ossher, H., and P. Tarr. 1999. *Multi-Dimensional Separation of Concerns in Hyperspace.* Technical Report RC 21452 (96717) 16APR99. Yorktown Heights, New York: IBM T. J. Watson Research Center.

Parnas, D. L. 1972. "On the Criteria to be used in Decomposing Systems into Modules." *Communications of the ACM* 15(12):1053–1058.

Pope, S. T. 1991. "Introduction to MODE: The Musical Object Development Environment." In S. T. Pope, ed. *The Well-Tempered Object: Musical Applications of Object-Oriented Software Technology.* Cambridge, Massachusetts: MIT Press, pp. 83–106.

Rumbaugh, J., et al. 1991. *Object-Oriented Modeling and Design.* Upper Saddle River, New Jersey: Prentice-Hall.

Schoenberg, A. 1967. *Fundamentals of Music Composition.* London: Faber and Faber.

Sloboda, J. A. 1985. *The Musical Mind: The Cognitive Psychology of Music.* London: Oxford University Press.

Spiegel, L. 1988. "Old-Fashioned Composing from the Inside Out: On Sounding Un-Digital on the Compositional Level." Paper presented at the 8th Symposium on Small Computers in the Arts, 2–5 November, Philadelphia.

Tarr, P., and H. Ossher. 2000. *Hyper/J User and Installation Manual.* Technical Report. Yorktown Heights, NY: IBM.

Xerox. 2002. *The AspectJ Programming Guide.* Technical Report. Stamford, Conneticut: Xerox Corporation.

Downloaded from http://direct.mit.edu/comj/article-pdf/31/4/47/1854942/comj.2007.31.4.47.pdf by guest on 17 September 2021