Symmetric Composition of Musical Concerns

Patrick Hill
The Open University
Walton Hall
Milton Keynes.
MK7 6AA

PatrickHill@bcs.org.uk

Simon Holland
The Open University
Walton Hall
Milton Keynes.
MK7 6AA

s.holland@open.ac.uk

Robin Laney
The Open University
Walton Hall
Milton Keynes.
MK7 6AA

r.c.laney@open.ac.uk

ABSTRACT
Aspect-oriented programming (AOP) describes a range of techniques that enable the separation, organisation and composition of various programming concerns that cannot be adequately encapsulated using the principal decomposition mechanisms available to modern programming languages.

Naturally, most AOP-related research is focussed on its application to the development of computer software. However, we believe that it is worthwhile considering whether AOP and cognate techniques might be usefully adapted as a means for an end-user to organise, represent and compose information in computer systems that support application domains in which scattering and tangling are present.

Music is notoriously rich in deeply tangled relationships. Moreover, there is no universally accepted representation of music that simultaneously represents all dimensions of interest to the composer.

In this paper we describe Aspect Oriented Music Representation, an approach to the organisation, representation and composition of musical materials based on MDSOC. Our approach extends MDSOC by adding a dynamic hyperspace and by allowing users to write detailed composition expressions using an extensible set of compositors. We introduce the concept of composition history, enabling symmetric composition to be related to joinpoints, demonstrating a way to combine symmetric and asymmetric aspect approaches.

Categories and Subject Descriptors
D.2.2 **[Software Engineering]**: Design Tools and Techniques - *Object-oriented design methods, Aspect-oriented design*; D.2.1 **[Software Engineering]:** Requirements / Specifications **-** *Methodologies, Separation of Concerns*; D.2.3 **[Software Engineering]:** Coding Tools and Techniques; D.3.3 **[Programming Languages]:** Language Constructs and Features - *Aspects;* J.5 **[Computer Applications]:** Performing Arts - *Music*

General Terms
Design, Languages, Human Factors.

Keywords
Aspect-oriented programming, Multi-dimensional Separation of Concerns, music composition, music representation.

1. INTRODUCTION
Aspect-Oriented Programming (AOP) [27] and cognate techniques [1, 31, 38] aim, in various ways, to assist in managing the organisation and composition of software components that appear at multiple, possibly unrelated, loci within the structure of a software system. Such elements are said to *crosscut* the system's structure. Approaches to AOP vary. Some approaches, such as Composition Filters [1] and Adaptive Programming [30], support the separation and composition of specific concern types. Other approaches, such as Hyper/J [50] , AspectJ [51], and Caesar [33] are of more general-purpose nature. Two key approaches to general-purpose AOP are symmetric and asymmetric [19]. In general, symmetric AOP, such as Multi-Dimensional Separation of Concerns (MDSOC) [38] , which is the main subject of this paper, is concerned with the identification and encapsulation of discrete software units that together represent the implementation of a particular concern. Symmetric approaches enable such units to be composed together to form complete software systems. In contrast, Asymmetric AOP implementations, such as AspectJ [51] and derivative dynamic aspects systems such as AspectS [25], largely consider the augmentation of a base program, through the application of encapsulated crosscutting concerns at specified points in a program's static structure, or dynamic execution graph.

Scattered through the AOSD literature are claims that the same ideas could be applied more widely than just to object-orientation [14], and indeed more widely than software concerns [38]. To the best of our knowledge, the present programme of work, alluded to in [21-23] and detailed here, is the first to apply ideas systematically from AOP to music representation and composition. An experiment in music notation and composition that appears to be very loosely inspired by AOP is mentioned in passing in [4].

In a similar way to software, problems of crosscutting appear to exist, at various levels, in music [21-23]. It is common experience that music is not simply a stream of random notes. Rather, when listening to music, we rely, to a great extent, upon the perception of repetition and variation of musical material [34]. Therefore, in composing a piece of music, composers tend to utilise a finite set of musical resources that are variously combined and transformed to achieve a balance between novelty and repetition and form a coherent musical whole [3, 28, 43-45]. The processes of musical composition can be usefully seen to involve the multidimensional scattering and tangling of musical ideas and transformational and combinatorial processes. At a level of abstraction, the perceived "musical surface" can be viewed as the result of the composer's tacit weaving together of a "tangled web" [11, 35] of musical structures and dimensions.

Music, as typically conceived of and represented, is multidimensional in that it contains multiple types of information, some of which are presented simultaneously. There are a number of musical dimensions, such as the principal perceptual dimensions of pitch, rhythm, loudness and timbre [32], which are inherent in music. Other dimensions may overlay these dimensions; harmony for example is a higher-level organisation of pitch. Additionally, musical information may be transformed or filtered to produce new, but related material. Transformation therefore presents another dimension. A music composer is also concerned with the ways in which musical structures are combined to form new structures. Moreover, structures across various simultaneously presented dimensions of musical concern may form polyarchies (overlapping hierarchies) of arbitrary depth [28].

Since the pioneering experiments of the 1950's [24], researchers and practitioners from diverse scientific and artistic communities have explored the application of computers to a varied range of musical activities. Composers have used and developed music-based programming language extensions to assist in the processes of musical composition [15, 37, 41]. The approach of each system influences its underlying musical ontology and the design of its musical representation. While computer systems generally demand a formal and precise representation of the data upon which they operate, the lack of consensus on what music is and how it is structured contributes to a "challenging" array of representational problems [10]. These problems are further complicated by the multidimensional nature of musical information and the requirement to represent task related viewpoints of musical data across the tangle of relationships that exist between various musical elements and dimensions.

While the composition of software systems, and musical works are clearly different, we suggest, that there are strong similarities between the kinds of crosscutting that occur in software structure, and those that occur in the structuring of musical materials. We believe, therefore, that it is worthwhile investigating ways in which Aspect Oriented techniques may be usefully adapted and applied to the computational representation of music for the purposes of musical composition.

Our approach involves developing new computer mediated tools to allow composers to deal more flexibly with cross-cutting musical concerns. Our approach starts with the application of AOSD ideas directly to musical materials and processes rather than their conventional application to software units such as methods and fields. This ultimately leads to musical tools in which, motivated by the same reasons as AOSD, musical materials are organised differently from existing musical applications and are embodied within software that deals explicitly with the separation and composition of concerns in the application domain. Moreover, the end-user of such applications interacts through the exposed "aspect-oriented" model.

In this paper, we describe a computational representation of music that facilitates the organisation of discrete musical ideas and their composition into higher-level units. In the first section we outline our general representation of musical information, and describe some of the ways in which new musical information may be obtained through the composition of a finite set of musical elements. In the second section we describe a domain-specific approach to the symmetric organisation of this musical representation, inspired by the work of Ossher et al [38]. The contributions of this paper are in applying aspect-oriented concepts to the concerns of an *application* domain, and exposing AOP to the end-user. We also suggest *composition history* as a way in which symmetric and asymmetric approaches may be usefully combined.

2. REPRESENTATION OF MUSICAL INFORMATION

In order to deal with crosscutting musical concerns, rather than the classes, methods and attributes that are the primitive elements of object-oriented programming concerns, we need to focus on ways of representing and weaving fundamental musical building blocks. In our representation, a *Music Unit* contains a single type of musical information, such as pitch or rhythm. Music Units are the lowest level of granularity that may be expressed using our approach. Of course, *reductio ad absurdam*, a music unit could contain a single pitch or rhythmic value, but we would argue that composers think in terms of higher level composites of such values, for example, a melodic line, cell, tone row, chord progression or rhythmic pattern [6, 7, 39, 44, 47, 48]. Either case can, without loss of generality, be represented using a sequence of values, since a sequence may contain just a single value. Consequently a music unit represents an *ordered sequence* of *ordered collections* of values. This enables simultaneous structures, such as chords, to be expressed without losing any ordering information between simultaneous elements. While not all simultaneous values may be meaningful, the approach does not impose any constraints. Moreover, the semantics of the musical values stored in a music unit are neither defined nor constrained by the approach.

All the values within all collections in the ordered sequence relate to a single *musical type*. For example, a Music Unit may contain a sequence of pitches or chords, a rhythmic pattern, or a sequence of musical transformations such as pitch transposition, rhythmic augmentation etc. A music unit may be populated by an algorithmic generative process, or from a data source, such as a MIDI file.

In general, musical types are not restricted to a single representation. For example, a music unit representing pitch may contain a mixture of values that represent pitch in a variety of ways, such as frequency, MIDI note number (an integer in the range 0-127), symbolic note name (eg, 'C4') etc. [11, 41]. However, for simplicity, in this paper we use consistent representations within a music unit type.

2.1 Composed Music Units

Music Units are not inherently associated with other Music Units. A Composed Music Unit (CMU) is a *container* [20] that has been formed by composing together particular Music Units. For example, pitch and rhythm music units might be composed together into a CMU that represents a melody. A Music Unit type appears only once in each CMU. Since CMUs contain Music Units, CMUs may be composed with other CMUs.

2.1.1 CMU Representation

Each CMU may be considered as containing a dictionary of indexable ordered collections, each collection representing a Music Unit. The dictionary key to each collection is the name of the Music Unit *type* that it represents. Each element of the ordered collection is accessible through an index (I) and, as described above, each element contains a collection of values (S).

In addition to musical information, a CMU may contain code fragments that are used to transform the musical information in some way. Transformations are represented as code fragments, in the music unit type #transform. In order to help visualise the representation, consider the following simple examples.

A populated CMU, containing only a #pitch music unit consisting of the notes of a C major triad (the notes C,E and G), occurring in sequence, is shown in Figure 1.

Type	Sequence			
#pitch	**I**	1	2	3
	S	‹C›	‹E›	‹G›

Figure 1. Pitches of C major triad in sequence

A populated CMU containing a single C Major triad, occurring simultaneously, is shown in Figure 2

Type	Sequence	
#pitch	**I**	1
	S	‹C,E,G›

Figure 2. Pitches of C major triad in unison

A populated CMU containing two music unit types, #pitch and #rhythm, respectively consisting of the notes of a C major triad occurring in sequence and a rhythmic pattern[1] is shown in Figure 3. Here numeric values in the #rhythm music unit type indicate the duration of a pitch. Note that because music units are correlated by index, a CMU represents a homorhythmic structure. In other words, CMUs do not support multiple simultaneous rhythmic figures.

[1] The detailed representation of pitches and rhythm values is outside the scope of this paper

Type	Sequence			
#pitch	**I**	1	2	3
	S	‹C›	‹E›	‹G›
#rhythm	**I**	1	2	3
	S	‹4›	‹8›	‹8›

Figure 3. Pitches of C major triad, with rhythmic values, in sequence

2.2 Composing CMUs

CMUs may be composed together to produce new CMUs. Since the term *composition* is heavily overloaded in the context of software and music, we refer to the process of composing the musical and programmatic data from selected CMUs into a single CMU as *weaving*. Processes that perform weaving are termed *weavers*.

2.2.1 Weaving

The weaving of CMUs is achieved by merging, according to the logic implemented by a *weaver*, the ordered collections of *matching* music unit types. Usually, music unit types are matched by name. Thus, for example, the information from the pitch type of one CMU is woven with the pitch information of another. However, in principle, this matching may be overridden. For example, a composer may wish a sequence of rhythmic values to be interpreted as pitch values. In this case, the #pitch music unit of one CMU might be woven with the #rhythm music unit of another.

The weaving of CMUs may be specified by a *composition expression* that specifies the CMUs to be woven, and the weavers required to achieve the desired weave. Two of the simplest, and possibly musically most useful, weavers are *sequential* and *parallel.*

2.2.1.1 Sequential Weaving

Sequential weaving appends to the *ordered collection* of each of the types being composed. Figure 4 shows two CMUs, CMU1 and CMU2 and the result of the sequential composition of these CMUs.

CMU1

Type	Sequence		
#pitch	**I**	1	2
	S	‹C›	‹D›

CMU2

Type	Sequence		
#pitch	**I**	1	2
	S	‹E›	‹F›

CMU1 and CMU2 composed in sequence

Type	Sequence				
#pitch	**I**	1	2	3	4
	S	‹C›	‹D›	‹E›	‹F›

Figure 4: Sequential Composition of CMUs

In sequential composition, all the music unit types that exist in the CMUs being sequentially woven together are represented in the resulting CMU. The example in Figure 5 shows the sequential weaving of two CMUs, CMU1 and CMU2, and that the result is actually of their *parallel* composition.

CMU1

Type	Sequence				
#pitch	I	1	2	3	4
	S	‹C›	‹D›	‹E›	‹F›

CMU2

Type	Sequence				
#rhythm	I	1	2	3	4
	S	‹1›	‹2›	‹2›	‹1›

CMU1 and CMU2 composed in sequence

Type	Sequence				
#pitch	I	1	2	3	4
	S	‹C›	‹D›	‹E›	‹F›
#rhythm	I	1	2	3	4
	S	‹1›	‹2›	‹2›	‹1›

Figure 5: Sequential Composition of multiple types

The semantics of sequential weaving are that the set of music unit types contained in the resultant CMU is the union of all music unit types in the composed CMUs. Each music unit type in the resultant CMU contains a concatenation of the sequences of the same music unit type from the composed CMUs. These relationships are shown formally in Figure 6.

If $C = A + B$ where *A* and *B* are CMUs and the symbol '+' represents sequential weaving, then the set of music unit types *T(C)* in the resultant CMU *C* is the union of the music unit types in A and B.

$$T(C) = T(A) \cup T(B)$$

And each music unit is composed sequentially

$$\forall x \in T(C):\ s(C,x) = s(A,x) \,\hat{}\, s(B,x)$$

where
s(c,t) is the sequence stored for music unit type *t* in CMU *c*.
^ indicates sequence concatenation.

Figure 6. Formal Expression of Sequential Weaving

There is no requirement for there to be the same number of elements in the ordered collections of different music unit types.

An interesting result is, therefore, that the composition of a pitch sequence with a rhythmic sequence may be achieved through *sequential* composition rather than the possibly more intuitive parallel composition.

2.2.1.2 Parallel Weaving

Parallel weaving correlates indexes between the collections of the music units being composed, and adds to the *collection* found at each index. Figure 7 shows two CMUs, CMU1 and CMU2, and the result of their parallel weaving.

CMU1

Type	Sequence			
#pitch	I	1	2	3
	S	‹C›	‹E›	‹G›

CMU2

Type	Sequence			
#pitch	I	1	2	3
	S	‹E›	‹G›	‹B›

CMU1 and CMU2 composed in parallel

Type	Sequence			
#pitch	I	1	2	3
	S	‹C,E›	‹E,G›	‹G,B›

Figure 7. Parallel Composition of CMUs

The semantics of parallel weaving are that the resultant CMU contains the union of the set of all music unit types in the composed CMUs. Each resultant music unit contains an indexed sequence of ordered collections. Each such collection contains the index-correlated set of values from the composed music units. This is shown formally in Figure 8.

If C = *A // B w*here A and B are CMUs, and the symbol '//' represents parallel composition, then the set of types T(C) in the resultant CMU *C* is the union of the types in A and B.

$$T(C) = T(A) \cup T(B)$$

And each type is composed in parallel such that

$$\forall x \in T(C),$$
$$\forall i\ 1 \le i \le \max(\#s(A,x), \#s(B,x)):$$
$$m(C,x,i) = m(A,x,i) \cup m(B,x,i)$$

where
T(c) is the set of type in CMU *c*
s(c,t) is the sequence stored for music unit type *t* in CMU *c*.
m(c,t,i) is the collection stored for music unit type *t* in CMU *c* at sequence index *i*
#s is the number of elements in the ordered sequence s.

Figure 8. Formalism for Parallel weaving

2.3 Applying Transformations

We have stated earlier that in addition to musical data, a CMU may contain transformational processes. The *value* of a CMU is obtained by executing each of the transformations found in its #transform music unit to yield a new CMU that contains no #transform type.

For example, a CMU containing a transformation and its evaluation are shown in Figure 9.

CMU1

Type	Sequence				
#pitch	**I**	1	2	3	4
	S	‹C›	‹D›	‹E›	‹F›
#rhythm	**I**	1	2	3	4
	S	‹2›	‹4›	‹4›	‹2›
#transform	**I**	1			
	S	‹R_AUGMENT (3)›			

Evaluation of CMU1

Type	Sequence				
#pitch	**I**	1	2	3	4
	S	‹C›	‹D›	‹E›	‹F›
#rhythm	**I**	1	2	3	4
	S	‹6›	‹12›	‹12›	‹6›

Figure 9. Evaluation of a CMU

The execution of the R_AUGMENT(3) transformation, in this example, causes the elements of the #rhythm music unit to be lengthened (augmented) by tripling their value.

If the #transform collection contains multiple transformations, then they are executed in sequence.

Transformations are not restricted to operating on a single type. Rather, transformations have unrestricted access to the entire CMU. Thus, for example, we might define a transform DROP_ALTERNATE that removes every other entry from *all* the music units in the CMU, as illustrated in Figure 10.

CMU1

Type	Sequence				
#pitch	**I**	1	2	3	4
	S	‹C›	‹D›	‹E›	‹F›
#rhythm	**I**	1	2	3	4
	S	‹2›	‹4›	‹4›	‹2›
#transform	**I**	1			
	S	‹R_AUGMENT (3), DROP_ALTERNATE›			

Evaluation of CMU1

Type	Sequence		
#pitch	**I**	1	3
	S	‹C›	‹E›
#rhythm	**I**	1	3
	S	‹6›	‹12›

Figure 10. Evaluation of multiple transformations

Musical composition processes, such as Counterpoint, Fugue [40] and Serial Composition [39], often involve musical transformations, such as the reversal of pitch and / or rhythm sequences. These kinds of transformation are easily implemented using this approach.

If a CMU contains no #transform collection, or an empty #transform collection, then the value of the CMU is the CMU itself.

In the remainder of this paper, we will notate the evaluation of a CMU C as @C.

2.4 Composing Units with Transformations

As discussed in 2.3, *evaluation* applies the transformations in the #transform collection to the woven elements from its component CMUs.

There are several possible results from composing CMUs that contain transformations, depending upon whether or not the CMUs are evaluated prior to composition.

To illustrate this, consider the composition of two CMUs, A and B in which CMU A contains a pitch sequence PA and a transformation TA and CMU B contains a pitch sequence PB and a transformation TB. Figures 11-17 show variations on the composition of CMU A and CMU B that can be achieved using evaluation.

Type	Sequence		
#pitch	**I**	1	2
	S	‹PA›	‹PB›
#transform	**I**	1	2
	S	‹TA›	‹TB›

Figure 11. Sequential Composition A + B

Type	Sequence		
#pitch	**I**	1	2
	S	‹TB(TA(PA)›	‹TB(TA(PB))›

Figure 12. Value of @(A + B)

Type	Sequence		
#pitch	**I**	1	2
	S	‹TA(PA)›	‹PB›
#transform	**I**	1	
	S	‹TB›	

Figure 13. Composed CMU (@A + B)

Type	Sequence		
#pitch	**Index**	1	2
	Set	‹TB(TA(PA)›	‹TB(PB))›

Figure 14. Value of (@A + B)

Type	Sequence		
#pitch	**I**	1	2
	S	‹PA›	‹TB(PB)›
#transform	**I**	1	
	S	‹TA›	

Figure 15. Composed CMU (A + @B)

Type	Sequence		
#pitch	**Index**	1	2
	Set	‹ TA(PA)›	‹TA(TB(PB))›

Figure 16. Value of (A + @B)

Type	Sequence		
#pitch	**I**	1	2
	S	‹TA(PA)›	‹TB(PB)›

Figure 17. Composed CMU @A + @B

To help make these ideas more concrete, consider the example implementations for the PA, PB, TA and TB units shown in Figure 18. The TRANSPOSE transformation shift pitches by an amount specified by its parameter. If the parameter is negative, then the pitch is shifted down.

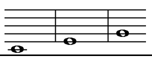
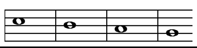
CMU	Unit	Implementation
A	PA	
	TA	TRANSPOSE(3)
B	PB	
	TB	TRANSPOSE(-7)

Figure 18. Example unit implementations.

As we discussed in the introduction, musical compositions often use several musical ideas that are variously combined. Figure 19 illustrates the values of CMUs woven with the various weaving expressions discussed above. Each CMU value represents a particular variation that may be obtained from the composition of the two CMUs A and B. Note however that various structural characteristics of the results remain constant. We can see, for instance, that a group three ascending pitches is always followed by a group of four descending pitches, and that the 'distance' between pitches in each group remains constant.

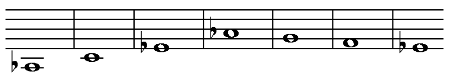
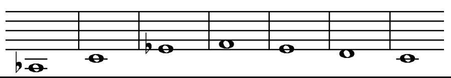
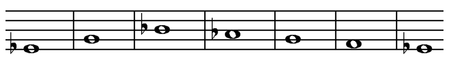
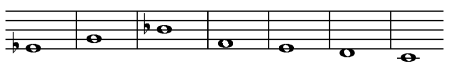
Weaving Expression	CMU Value
@(A + B)	
@A + B	
A + @B	
@A + @B	

Figure 19. Weaving results

3. ORGANISING MUSICAL MATERIALS

The foregoing has introduced the concept of Composed Music Units as composable containers that aggregate musical and transformational information. However, each CMU exists in isolation; it is not possible to specify a relationship to any particular musical dimension or concern**.** One of the main difficulties in building music representations is that there is no single, all-encompassing, musical ontology. Rather, particular music systems impose their own ontology on the composer.

A similar problem exists in software engineering. In object-oriented software development, for example, the unit of decomposition, namely the Class, is imposed upon the software developer as the fundamental structuring mechanism. Multi-Dimensional Separation of Concerns [8] is an AOP technique that enables software code fragments to be logically related. Software systems may then be composed by reference to these logical groupings. Thus the dominance of the class decomposition is overcome.

Like Subject Oriented Programming (SOP) [17] and Mixins [36], MDSOC, as applied to software, is concerned with the separation and composition of program fragments [9]. Similarly, in our work, we want to be able to compose fragments containing both multidimensional musical data and algorithmic transformations into higher-level components. Moreover, we want to be able to *organise* the fragments and composed units without imposing any particular musical ontology.

Non-musicians might wonder why musicians would want to do this. Aspect Oriented Music Representation (AOMR) offers composers capabilities not easily obtainable otherwise. These capabilities lie particularly in rapidly carrying out and fine-tuning wide-ranging musical "what if" experiments. Existing tools allow composers to perform limited what-if experiments. By contrast, AOMR allows musical what-if experiments to range over any musical dimensions or concerns, and to cross-cut low level details in one dimension with high level abstractions in another. Changes can be applied over arbitrarily specifiable scopes - not just scopes based on temporal intervals or voices. This is particularly useful, because while music is experienced in time, it is generally not composed in a left-to-right fashion. Composers seem to work with different levels of abstraction simultaneously and with incomplete ideas in various musical dimensions [49]. The notion of thematic unity means that composers are often very economical about the materials they use, but deeply concerned with exploring interesting ways in which they can be combined and transformed. Musicians also tend to *problem seek* rather than just problem solve [26] which makes what-if experiments, whether tacit or explicit, vital. AOMR may not be well suited for all musical tasks, any more than AOP for all programming, but it offers advantages not readily available otherwise.

We have therefore used MDSOC as a basis for our approach. There are, however, some distinctive requirements of musical composition. Firstly, while the ordering of composition in software is sometimes irrelevant, it must be possible to exert precise control over the ordering of composition of music units of the corresponding types. Secondly, unlike the MDSOC approach to software composition, (music) units may appear multiple times within a composition. Finally, units may be transformed as part of the composition.

In this section, we describe our extensions to MDSOC for use in composing CMUs. We stress that it is not anticipated that the end-user of our system will interact directly with the MDSOC-like formalisms described in this section. Rather, the representation described here will be used internally within our system. The user will be presented with a UI that facilitates the construction of the artefacts described.

3.1 Hyperspaces

Our approach to organising musical materials is based upon the MDSOC approach to software composition [38]. In a similar way to MDSOC, in our approach, discrete CMUs are organised into a *hyperspace*. Each unit is named and associated with a *dimension* and a *concern in that dimension*. The dimensions and their concerns that are contained by the hyperspace are entirely arbitrary, enabling the composer to classify each unit according to their particular structuring preferences. Each unit implementation appears only once in the hyperspace.

In the following examples, we use the Hyperspace shown in Figure 20. For simplicity of exposition, each unit in this hyperspace contains only a single type, though in general, this is not the case.

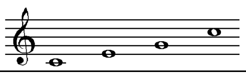
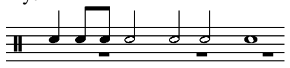
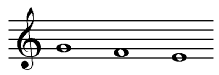
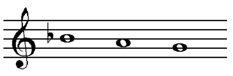
Dimension	Concern	Unit	Unit Implementation
Phrase1	Melody	A	#pitch 
Phrase1	Rhythm	R	#rhythm 
Phrase1	Melody	B	#pitch 
Phrase2	Melody	B	#pitch 

Figure 20. Example Hyperspace

3.2 Hyperslices

A hyperslice is an abstract slice through the hyperspace and is expressed as set of regular expressions that may be used to match *dimension.concern.*

For example, the hyperslice specification

“.*”.”Melody”

matches all units from all dimensions that are in the “Melody” concern. In our example hyperspace, this would yield three CMUs; Phrase1.Melody.A, Phrase1.Melody.B and Phrase2.Melody.B

3.3 Hypermodules

A hypermodule is a specification that is used to construct, through weaving, a new CMU from units within the hyperspace.

For example, consider a CMU that represents the musical snippet shown in Figure 21.

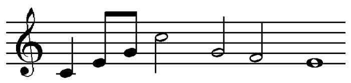


Figure 21. A musical snippet

By referring to the hyperspace in figure 21, such a CMU might be composed by the following hypermodule specification

```
Musicunit: Intro1
Hyperslices: Phrase1.*
Relationships:
  mergeByName;
Composition:
  ".*A" + ".*B" + ".*R";
```

This hypermodule specification composes a CMU called “Intro1”, as specified in the Musicunit specification.

The Hyperslice specification defines the hyperslices from which the CMUs to be composed will be drawn.

The Relationships specification specifies the general composition strategy. The composition strategy describes how names will be matched in the composition expression. The “mergeByName” composition strategy used here indicates that units drawn from the hyperslice are to correspond to those in the composition expression if the names of the units match.

The Composition specification contains an expression that defines *how* the CMUs are to be woven together. The expression contains CMU names to be matched from the hyperslice and operators representing weavers. For example, ‘+’ represents a sequential weaver, while ‘//’ represents a parallel weaver.

In this example, the composition expression A + B + R is resolved as follows:

The hyperslice specification includes only those units in the Phrase1 dimension. Namely

Phrase1.Melody.A
Phrase1.Melody.B
and Phrase1.Rhythm.R.

Matching these against the composition expression

```
".*A" + ".*B" + ".*R"
```

yields

```
Phrase1.Melody.A + Phrase1.Melody.B +
Phrase1.Rhythm.R
```

A second CMU may be composed using the following hypermodule specification

```
Musicunit: Intro2
Hyperslices: Phrase1.*, Phrase2.*
Relationships:
      overrideByName;
Composition:
  ".*A" + ".*B" + ".*R";
```

In this specification, the overrideByName relationship indicates that in the event that multiple matches are found for a unit in the hyperslice, then the *last* found unit should be used. The search order is dictated by the order in which hyperslices are specified.

Consequently, the composition expression in this hypermodule is resolved as follows:

The hyperslice includes all units in all concerns from the Phrase1 and Phrase2 dimensions, in this order. Namely

Phrase1.Melody.A,
Phrase1.Melody.B,
Phrase1.Rhythm.R
and Phrase2.Melody.B.

“*.A” can only be matched by Phrase1.Melody.A

"*.B" can be matched by Phrase1.Melody.A *and* Phrase2.Melody.B
"*.R" can be matched only by Phrase1.Rhythm.R

The ambiguity of matching "*.B" is resolved by the overrideByName relationship. This relationship indicates that the last found match is used. In this case, the match is Phrase2.Melody.B.

Thus the composition expression is resolved to:

```
Phrase1.Melody.A  +  Phrase2.Melody.B  +
Phrase1.Rhythm.R
```

Since we currently assume that the semantics of the CMU are such that the dimensions of each note, in this case pitch and rhythm, are obtained by correlating indices across all composed dimensions, then this composition yields the musical fragment shown in Figure 22.

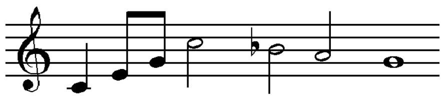


Figure 22

3.4 Expressing Relationships Between CMUs

In musical composition, often a particular musical idea may be transformed and the transformed version used in composing new musical materials. This results in the formation of so-called *evolutionary chains* [8]. Supporting this type of composition requires the ability to define new CMUs *and* add them to the hyperspace. Thus the hyperspace is a dynamic structure that evolves with the musical composition. In particular, the composer may not necessarily separate out all the dimensions and concerns *ab initio*, but may subsequently wish to remodularise a unit by separating its concerns as the musical work evolves.

For example, by inspection, we can see that Phrase2.Melody.B is actually a transposition, 3 semitones up, of Phrase1.Melody.B

Rather than defining Phrase2.Melody.B in terms of absolute musical data, we can define a CMU that expresses this relationship.

In this example, a transformation unit Transpose has been added to the hyperspace in the Transforms.Pitch dimension and concern.

The Transpose transformation unit takes as an argument the number of semitones to transpose.

```
Musicunit: Phrase2.Melody.B
Hyperslices: Phrase1.Melody,
Transforms.Pitch
Relationships:  MergeByName;
Composition:  Transpose(3) + B;
```

The composition expression is resolved to:

```
Transforms.Pitch.Transpose(3) +
Phrase1.Melody.B
```

By providing a fully qualified music unit specification (Phrase2.Melody.B), the newly composed unit may be added to the hyperspace in its own dimension and concern.

3.5 Multiple Matches

The `mergeByName` relationship has the possibility to return multiple matching results.

For example, the hyperslice specification

Phrase1.Melody, Phrase2.Melody

results in two matches for the unit B, namely Phrase1.Melody.B and Phrase2.Melody.B.

By default, the `mergeByName` relationship causes matching units to be woven in sequence, in the order in which they are found.

Thus, given the hyperslice specification shown above, the composition expression Transpose(3) + B would be resolved to

Transpose(3) + (Phrase1.Melody.B + Phrase2.Melody.B)

If the default sequential weaving is not required, then the composition expression must explicitly state the desired weaving such that all ambiguities are resolved.

3.6 Other Weavers

Unlike the composition of software, through an approach such as MDSOC, the *ways* in which musical information may be composed together are open-ended. The two basic weavers, sequential and parallel, described above are useful in musical composition. However, other weavers might be required to perform specific weaving operations, such as the appoggiatura composition described in [12]. Briefly, an appoggiatura is a "time-taking" musical "ornament" O that is associated with, and prefixes, a main musical structure T. If T is placed in sequence, following a structure S, then the appoggiatura O associated with T must *coincide* with S if T is to immediately follow S. Consequently, unlike simple sequential weaving, an appoggiatura-style sequential weaving of two units must *modify* the first unit to include the appoggiatura of the second. Unlike sequential and parallel weaving, an important requirement of this kind weaving is that the weaver itself knows how to interpret the information contained within the music units being composed.

3.7 Composition History

The symmetric organisation and composition of musical materials described above is partial; it is not expected that entire compositions will be structured using this approach. Rather, CMUs will be assembled in time through an approach, called MusicSpace, which incorporates an analog of Asymmetric AOP. Full description of this approach is outside the scope of the present paper, but briefly, MusicSpace supports temporal joinpoints. At each joinpoint it is possible, through pointcut expressions, to query the content of the musical information that is to be rendered at that point in time.

As observed in [5], composed units have no recollection of their components once they are composed. Therefore, in order to support richly expressive pointcuts within MusicSpace, the mechanism through which CMUs are composed produces, for each element of each sequence of each collection, a structure that identifies the dimension, concern and unit name of all units that have created or affected the element. Thus it is possible, for example, to define pointcut expressions that identify all pitches that have been generated from a particular unit, even though the transformational processes through which the note has passed may make such an analysis difficult, or impossible, from the musical surface alone. This, we claim, has applications, not only to musical composition, but also to analytical and pedagogical music systems.

4. RELATED WORK

There are many approaches to, and implementations of, AOP [1, 19, 25, 27, 30, 31, 33, 38, 50, 51] and, as noted in [42], different types of crosscutting concern may be better handled by one approach over another. Our work, as reported in this paper considers the static composition of units from components representing different concerns, and as such, our primary influence is MDSOC [38]. However, rather than applying the ideas to classes, methods and attributes, we have applied them to domain concepts; musical materials, musical processes and transformations and domain-specific forms of weaving. Thus, at the detailed level, our work borrows ideas from the music representation literature [2, 10, 46].

Approaches to musical representation are diverse. Some approaches, such as [13, 15, 29] support the separation of concerns for specific domains of musical interest, such as harmony, or stylistic concerns. However our research considers more general musical applications. The CHARM representation [46] abstracts musical events from structure, enabling various structures to be superimposed over a single set of events. However, CHARM considers an event as a composite of musical dimensions, such as pitch and rhythm. The composition expressions that we have described are related to those of Music Structures [2]; a declarative representation that aims to model temporal and hierarchical musical relationships. However, Music Structures lacks a systematic repository for organising discrete musical elements. Consequently, there is a tight coupling between the composition specification and particular musical element instances.

We have suggested music composition tools could benefit greatly from both symmetric and asymmetric approaches, This requirement for multiple AOP approaches in software is supported by the Concern Manipulation Environment (CME) [18-20].

Finally, our work is in the spirit of Universal Composition [16], with each composed unit being built up of units that are stated only once, and whose composition is parameterised through a composition specification.

5. CONCLUSIONS

Since the inception of AOSD, there have been largely unsupported claims that the same ideas could be applied more generally than to object-orientation [14], and more widely than to software development [38]. To the best of our knowledge, the present programme of work is the first to investigate this claim in detail, using music representation and composition as a vehicle. A key step in our approach is to apply AOSD ideas directly to musical materials and processes rather than their conventional application to software units such as methods and fields.

Our approach enables the composition of multiple musical dimensions into higher-level composites, and for these composites to be themselves made available for further composition through the manipulation of a dynamic hyperspace. Composition expressions permit the definition of abstract orderings between the units being composed and enable the use of multiple weaving strategies within a given composition. The use of hyperslices and name matching abstracts the composition expression from the particular units being composed, while preserving structural relationships with respect to the hyperslice. Moreover, the weaving processes themselves are abstracted and extensible.

While this paper has described only the symmetric composition of musical information, we assert that *both* symmetric and asymmetric approaches are required for a full musical representation. We have proposed the use of a history of symmetric composition, which is made available for query at asymmetric-style joinpoints, as way to link both approaches

Thus, given the demanding multidimensional, temporally sequenced, polyarchic nature of music and music representation, applying MDSOC to music representation problems has demanded extensions of MDSOC. These extensions may be to some degree more generally applicable, and exportable back to MDSOC in its original domain. In summary, our approach extends MDSOC by adding a dynamic hyperspace and allowing users to write detailed composition expressions using an extensible set of compositors. In addition we introduced the concept of composition history, enabling symmetric composition to be related to joinpoints, demonstrating a way to combine symmetric and asymmetric aspect approaches at a high level of granularity.

5. REFERENCES

1. Aksit, M. and Tekinerdogan, B. Solving the modelling problems of object-oriented languages by composing multiple aspects using composition filters *Aspect Oriented Programming Workshop, ECOOP*, 1998.
2. Balaban, M. Music Structures: Interleaving the Temporal and Hierarchical Aspects in Music. in *Understanding Music with AI*, MIT Press, 1992.
3. Belkin, A. *A Practical Guide to Musical Composition.* http://www.musique.umontreal.ca/personnel/Belkin/bk/, 1995-1999.
4. Brown, S.S. and Robinson., P., Transforming musical notations for universal access. in *Designing a More Inclusive World: proceedings of the Cambridge Workshop on Universal Access and Assistive Technology*, (2004), pp 123-132.
5. Chitchyan, R. and Sommerville, I., AOP and Reflection for Dynamic Hyperslices. in *ECOOP'04 Workshop on Reflection,AOP, and Meta-Data for Software Evolution*, (Oslo, 2004), 29-35.

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

7. Cope, D. A Computer Model of Music Composition. in *Machine Models of Music*, MIT Press, 1993.

8. Crochemore, M., Iliopoulos, C.S. and Pinzon, Y.J. Computing Evolutionary Chains in Musical Sequences. *The Electronic Journal of Combinatorics*, *8* (2).

9. Czarnecki, K. and Eisenecker, U.W. *Generative Programming. Methods, Tools and Applications*. Addison Wesley, 2000.

10. Dannenberg, R.B. Music Representations Issues, Techniques and Systems. *Computer Music Journal*, *17* (3). 20-30.

11. Dannenberg, R.B., Desain, P. and Honing, H. Programming Language Design for Music. in De Poli, G., Picialli, A., Pope, S.T. and Roads, C. eds. *Musical Signal Processing.*, Swets & Zeitlinger., 1997, 271-315.

12. Desain, P. and Honing, H. Towards a Calculus for Expressive Timing in Music. *Computers in Music Research*, *3*. 42-120.

13. Ebcioglu, K. An Expert System for Harmonizing Chorales in the Style of J.S. Bach". in *Understanding Music with AI, Perspectives on Music Congnition*, MIT Press, 1992.

14. Filman, R. and Friedman, D.P. Aspect-Oriented Programming is Quantification and Obliviousness *Workshop on Advanced Separation of Concerns, OOPSLA*, Minneapolis, 2000.

15. Fry, C. Flavors Band: A Language for Specifying Musical Style. in Pope, S.T. ed. *The Well-Tempered Object: Musical Applications of Object-Oriented Software Technology*, MIT Press, 1991.

16. Gedenryd, H. *Universal Composition - An Optimal Scheme for Structuring (Software) Systems.* www.arcavia.com/cache/UCpaper.pdf. 2001.

17. Harrison, W. and Ossher, H. Subject-oriented programming: a critique of pure objects *Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, ACM Press, Washington, D.C., United States, 1993.

18. Harrison, W., Ossher, H., Sutton, S. and Tarr, P. *Concern Modeling in the Concern Manipulation Environment.* RC23344 (W0409-136). IBM. 2004.

19. Harrison, W., Ossher, H. and Tarr, P. *Asymmetrically vs. Symmetrically Organized Paradigms for Software Composition.* RC22685 (W0212-147). IBM. 2002.

20. Harrison, W., Ossher, H. and Tarr, P. *Concepts for Describing Composition of Software Artifacts.* RC23345 (W0409-140). IBM Research Division. 2004.

21. Hill, P., Holland, S. and Laney, R. Applying Aspect Oriented Programming to Music Computing *Proceedings Sound & Music Computing Conference (SMC04)*, Paris, France, 2004.

22. Hill, P., Holland, S. and Laney, R. *Using Aspects to Help Composers.* TR 2003/21. Department of Computing, The Open University. 2003.

23. Hill, P., Holland, S. and Laney, R. Using Dynamic Aspects in Music Composition *Dynamic Aspects Workshop, AOSD'04 International Conference on Aspect-Oriented Software Development*, Lancaster UK, 2004, 89-97.

24. Hiller, L. and Isaacson, L. Musical Composition with a High-Speed Digital Computer. in Schwanauer, S.M. and Levitt, D.A. eds. *Machine Models of Music*, MIT Press, 1958.

25. Hirschfeld, R. *Aspect-Oriented Programming with AspectS.* DoCoMo Communications Laboratories Europe. 2002.

26. Holland, S. Artificial Intelligence in Music Education: A Critical Review. in Miranda, E.R. ed. *Readings in Music and Artificial Intelligence*, Harwood Academic Publishers, 2000, 239-274.

27. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M. and Irwin, J. Aspect-Oriented Programming *European Conference on Object-Oriented Programming*, Springer-Verlag, 1997.

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

29. Levitt, D.A. Musical Dialects and Language.

30. Lieberherr, K.J. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns.* PWS Publishing Company, Boston, 1996.

31. Lieberherr, K.J., Silva-Lepe, I. and Xiao, C. *Adaptive Object-Oriented Programming using Graph Customisation.* College of Computer Science, Northeastern University. 1994.

32. Loy, G. and Abbott, C. Programming Languages for Computer Music Synthesis, Performance and Composition. *ACM Computing Surveys*, *17 No 2*.

33. Mezini, M. and Ostermann, K. Conquering Aspects with Caesar. *Proceedings AOSD*, 2003.

34. Minsky, M. Music, Mind and Meaning. in Schwanauer, S.M. and Levitt, D.A. eds. *Machine Models of Music*, MIT Press, 1993.

35. Miranda, E.R. *Composing Music with Computers.* Focal Press, 2001.

36. Moon, D. Object Oriented Programming with Flavors. *ACM SIGPLAN Notices, Conference proceedings on Object-oriented programming systems, languages and applications*, *21* (11).

37. Oppenheim, D.V. *DMix - A Multi Faceted Environment for Composing and Performing Computer Music: its Design, Philosophy, and Implementation.* Center for Computer Research in Music and Acoustics (CCRMA). 1992.

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

39. Perle, G. *Serial Composition and Atonality.* University of California Press, 1991.

40. Piston, W. *Counterpoint*. W.W. Norton & Company, 1947.

41. Pope, S.T. Introduction to MODE: The Musical Object Development Environment. in Pope, S.T. ed. *The Well-Tempered Object: Musical Applications of Object-Oriented Software Technology*, MIT Press, 1991.

42. Rashid, A. A Hybrid Approach to Separation of Concerns: The Story of SADES. *Lecture Notes in Computer Science*, *2192*. 231.

43. Russo, W., Ainis, J. and Stevenson, D. *Composing Music A New Approach*. University of Chicago Press, 1980.

44. Schoenberg, A. (ed.), *Fundamentals of Music Composition*. Faber and Faber, 1967.

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

46. Smaill, A., Wiggins, G.A. and Miranda, E.R. Music Representation - between the musician and the computer. in Smith, M., Smaill, A. and Wiggins, G. eds. *Music Education - An Artificial Intelligence Perspective*, Springer-Verlag, London, 1994, 108-122.

47. Smoliar, S. Process Structuring and Music Theory. in Schwanauer, S.M. and Levitt, D.A. eds. *Machine Models of Music*, MIT Press, 1993.

48. Spiegel, L. *Manipulations of Musical Patterns.* http://retiary.org/ls/writings/musical_manip.html. 1981.

49. Spiegel, L. Old Fashioned Composing from the Inside Out: On Sounding Un-Digital on the Compositional Level *8th Symposium on Small Computers in the Arts*, 1988.

50. Tarr, P. and Ossher, H. *Hyper/J*^{TM} *User and Installation Manual.* IBM Research. 2000.

51. Xerox. *The AspectJ Programming Guide.* Xerox Corporation. 1998-2002.