Complex Musical Pattern Description in Common Music

Reinhard Taube
Zentrum für Kunst und Medientechnologie
günterstraße 42 76137 Karlsruhe Germany
Email: hol@zkm.de

Abstract
Pattern description is a fundamental component of music composition. Composers utilize patterns at all levels of a musical composition, from deep structural organization to transitory surface phenomena. The computer can be a particularly effective tool for defining and exploring complex musical patterns. Common Music, an object-oriented composition environment, provides a facility for describing complex, dynamic patterns based on a class of object called item-stream. This paper provides a brief overview of item streams and describes recent extensions from its initial description in Computer Music Journal 15/2.

Introduction
The stream is a natural metaphor for thinking about musical patterns. Streams are sources of material and imply a dynamic, directional flow, just as music itself flows through time. An item stream is an object that represents a musical pattern as a typed stream of data. Item streams are generic resources that may be used in a variety of musical contexts: as tools for developing and exploring source material, as producers of parametric data in algorithmic composition, as structural organizers for sections of music and so on. Each item stream generates data according to a pattern type. The precise interpretation of data and type of pattern produced by an item stream are defined through multiple inheritance from two orthogonal superclass hierarchies: a taxonomy of data types and a taxonomy of pattern types. These superclasses also provide external control variables that permit composers to specialize characteristics on a per-stream instance basis, appropriate to the data and pattern type implemented by the stream.

As a basic rule, data supplied to a pattern or to control variables may be constant values or item streams of a compatible type. The ability to replace constant information with item streams of information enhances the power of the representation, allows both data and runtime characteristics of the stream to evolve in dynamic and complex ways, and provides the basic mechanism for building complex patterns out of a set of primitive pattern types.

Pattern Definition
A simple pattern is defined if an item stream contains only constant data. For example, integer data in a palindromic-interval-stream produces a stream of relative scale motions in simple palindromic order. A complex pattern is defined when an item stream contains one or more substreams in its pattern data. An embedded stream serves to delimit a subpattern within the superior stream's pattern.

By defining patterns in terms of subpatterns, a modest number of basic pattern types becomes, in effect, an open ended toolbox for assembling complex, hybrid streams that simultaneously share characteristics of several different pattern types. Since the depth and breadth pattern definition is open-ended, the description and behavior of a complex patterns must be left to the imagination of the composer, but cycles of randomness, random cycles of rotations, rotations of cycles and heaps, heaps of graphs, rotations of graphs, cycles of rotations of graphs, and so on are all possible. The ability to place patterns inside other patterns raises an important issue: how does a superior pattern decide when a subpattern is "finished" such that it can increment to the next element in its pattern?

Pattern Periods
All item streams organize data into discrete periods regardless of their pattern type. A period is just an
arbitrary grouping in the data returned from an item stream. The length of a period might correspond in some organic way to a pattern but it can also be constant, random, or evolve according to its own pattern rules. The main purpose of the period is to delimit the (current) extent of a subpattern in its superior's pattern. A superior stream is free to increment to the next element of its pattern only when the current element reaches the end of its current period. Constant data are defined to have period length 1. The end-of-period state signals a substream to calculate a new period length in anticipation of the next time it is (possibly) selected by the superior streams pattern. The current period length of an item stream can be any non-negative integer, including zero. The default period length is the sum of all periods in the data, which causes patterns to evolve in a "depth first" order. Conversely, if all subperiods are 1, subpatterns will be visited in a "breadth first" order. If a substream period is set to 0 it is "disappears" from its superior's pattern until it is (possibly) encountered again with a period length greater than 0. Since superpatterns can appear and disappear at will inside a superior pattern, the zero period length will cause even simple types of deterministic pattern selection to exhibit fairly complex behavior.

Item Stream Pattern Types

A taxonomy of pattern classes provides the set of properties necessary to define an item stream's pattern characteristics. There are currently six broad classes of pattern families: serial, random, accumulation, rotational, graph, and functional. A number of these classes are further specialized into more specific behaviors. All serial pattern types visit data sequentially, but vary in their specifics: a cycle continually loops over its data, sequence sticks on the last element, palindrome oscillates between the first and last element. The random pattern implements pseudo-random selection. Each item in a basic random pattern may have a probability weight and min/max constraints on consecutive selection. These constraints may be expressed as constant or dynamic values. The heap random pattern inherits from both random and serial. It shuffles data such that ordering is random but all elements are visited before any is repeated. The rotation pattern implements generic element swapping, with independent control over the first, last, stepping increment and swapping width for each rotational change. A change-ring subclass implements various sorts of rotation schemes invented by English bell ringers for composing long sequences of non-repetitive bell changes. The functional pattern is an escape to procedural description and provides the mechanism for implementing a number of pattern modifiers such as mirror, retrograde and repeat. The graph pattern implements a generic transition set utility. Pattern nodes are visited according to rules associated with each node. A transition rule may be expressed as a pattern, a probability table [Dodge and Jerse, 1985] or an arbitrary expression to evaluate. Probability tables can be used in conjunction with the graph's memory of past choices to implement higher order Markov selection. Since each node has its own transition rule, a single graph is free to implement hybrid transitions schemes, for example, some nodes might use Markov transition tables while other nodes select successors based on their own unique patterns.

Item Stream Data Types

Item streams are typed with respect to the data they hold. Type item denotes any Lisp object, so the most basic class of item stream is capable of holding anything that Lisp can represent. Since sequences are often work with sound parameter information, a number of item subtypes restrict or interpret data in purely musical contexts. These subtypes include rhythm for managing time in terms of tempo and beat, amplitude for symbolic or relative amplitude values, note, degree, and pitch for symbolic, keyname, or Hz representations of absolute frequency, and intervals, series and step for describing relative interval motion within a scale. The three interval type streams permit pattern offsets to be specified as control variables such that relative motion is mapped to absolute keyname, note name or Hz values. As with all control variables the offset may be expressed as a constant value, or pattern of offsets. Interval offsets can also be linked to the current value of an arbitrary frequency stream. This is
useful, for example, in algorithmic composition, to link independent and concurrently executing algorithms such that one algorithm describes its frequency as a pattern relative to the other algorithm.

Lisp Evaluation and Item Streams

Data supplied to a stream are normally evaluated once, when the stream is created. The means that, although the ordering of data changes dynamically according to the pattern type, the data values are treated as constants as they are returned by the stream. But it is sometimes useful to evaluate data as it is read from a stream. The expr class of item provides an efficient and automatic evaluation process for item stream data. An expr item represents a "promise" to evaluate its datum whenever the item is selected in the pattern. The datum itself may be any lisp expression. There are typically three situations where an expr is useful: to incorporate variables or lisp expressions into stream data with the expectation that their values rather than the expressions themselves will be returned by the pattern; to implement a "stream of streams"; in which objects returned by a pattern are other stream instances rather than their data; and to delay the specification of pattern data later than the time at which the stream that holds the data was created. The last case is particularly interesting since the stream then responds to conditions that did not exist when the stream was actually defined. Example 1 shows a random pattern whose probability weights change as a function of output time, after the stream itself has been created.

Example 1. The probability of the second node in this random pattern is expressed dynamically as a function of output time. Both nodes in the pattern are themselves random substreams with equal probabilities in their data, but the relative likelihood of picking from node (c e g e) ranges linearly from .25 at time 0 to 4 at time 25.

\[
\text{random}((d f b) \text{ weight: 1})
\]
\[
\text{random}((c a g e) \text{ weight: expr (interpolate (time, (0 .25 25 4))))}
\]

Multidimensional musical patterns

A single data type normally corresponds to a single attribute, or dimension, of sound. For example, a note duration corresponds to the frequency in a series of sound events. Musical motives and gestures, on the other hand, typically involve signatures over more than one attribute of sound. A pattern representation that supports only single-valued data cannot control multiple sound attributes within a single pattern stream. A solution for simple cyclic patterns in earlier languages such as PLA (Schottstaedt, 1983) was to control multiple attributes in parallel streams of data. But this solution not only lies outside the representation, gestural patterns involving random degrees of freedom cannot be implemented because attribute values that must remain concurrent across parallel streams are no longer guaranteed to be so. The problem of concurrent values in random ordering is solved if the pattern representation supports composite data types. By linking two or more attributes in a "multiple item" definition, a single pattern, regardless of its complexity, can control those attributes in parallel, as a single datum. Multiple items also serve as useful level of abstraction since the relationships between the attributes may be referred to as a named entity. Since each component of a new multiple item is itself a type, its component data may be represented by a constant values or streams of data of that type within the single multiple item. In other words, multiple

Music Representation, Data Structures

ICMC Proceedings 1994
items permit the definition of a stream of composite material in which each composite unit may itself involve streams of data.

Example 2. A cyclic general pattern defined across 2-asessions: frequency, rhythm and amplitude. The second and third genres are randomized to provide slight variations each time they are encountered. Three periods of the stream are depicted. Dotted lines define the individual genres.

defines-item gesture(note rhythm amplitude):
cycle(  
gesture: cycle(adf cs rect), cycle(16, f, 4), kf),  
gesture: (beap(fg ftb fnf bsf), cycle:beap(d, 4, 4).w., p1),  
gesture: cycle(15, f, 4, ed), beap(8, 4), cycle(p, mp, mf, f))

Bibliography