Stella: persistent score representation in Common Music

Heinrich Taube
Zentrum für kunst und Medientechnologie
Ritterstrasse 43, 7500 Karlsruhe 1, Germany
hki@zkdm.de, hki@ccrm.stanford.edu

ABSTRACT

Common Music (Taube, 1991) is a score description language that views musical composition as a procedural activity in which music is realized through the evaluation of independent compositional agents, called score parts. However, the activity of composition itself is often a state external, or literal process in which a composer shapes and develops "persistent" musical material over time until a completed composition emerges. This paper describes Stella, a representation system for persistent score information in Common Music. Stella is a super-set of Common Music's current representational scheme and is designed to simultaneously support both procedural (or algorithmic) and literal compositional styles. Stella provides flexible, open-ended data abstractions for the representation of musical material, a number of generic editing operators, and a top-level interpreter shell for command driven interaction. A graphic interface is planned for the SGI Iris Indigo. Stella is implemented in Common Lisp ( Steele, 1990) and CLOS (Bobrow, 1988) and runs on most standard hardware platforms.

BACKGROUND AND OVERVIEW

Common Music is a score description language that is designed to generate musical output in a variety of formats. Though its output is variable, Common Music's internal methodology is not; its design represents music composition as a procedural activity in which a composition, or score, is composed by a collection of independent processes called score parts. In this model, score and parts are used at temporary resources for the composition of the musical events in a composition, once musical output has concluded these objects are no longer needed and are reclaimed by the system's resource manager for later reuse. Thus, though Common Music's paradigm is a good model of "composition as process", it is not necessarily a good model of the "process of composition"! Though procedural composition is powerful, composers often want to work in a more textual manner in which musical events are, in some sense, similar to words or text, and in which persistent musical material is "pushed around", transposed, transcribed, replaced, revoiced, etc. Until a composition is finally completed. While procedural composition is elegant, not many composers cease their composition without wanting to stop on occasion (well ok, maybe if your name is Haydn,...) to revise, compare, re-examine, re-work earlier or following sections, unless the composer is explicitly interested in the generative process itself. The basic motivation behind Stella is to provide Common Music with a representation that permits both literary and algorithmic styles of composition to exist simultaneously. The composer is free to "move around" in a composition, working on this section or that, sometimes performing editing operations such as copy/paste, add/remove, shuffle, transpose, etc., sometimes running algorithms -- whatever is most appropriate for a particular situation. Several classes of algorithmic object are provided, the basic class will generate new material each time it is requested, a specialized type will remember what was last transposed and only generate new material if the current material is rejected by the composer. (This functionality permits, for example, algorithmic sketching utilizing random processes to be fished or untraced depending on the effectiveness of the result.) Elements in a score may be temporarily marked as "hidden", in which case they are skipped during the score evaluation process. Hiding and unhiding elements permits sketching alterations to be eased, musical to be slid, etc., without actually modifying actual structure in the composition. Stella also implements a notion of sticky, or default musical material. Sticky material is musical data that is merged with actual data to produce fully specified musical output events. Once a default is retrieved it remains in effect until replaced or overridden. Alternatively, sticky data can be used to perform Music Kit (Jaffe, 1991) style update events or to set global instrument defaults if working with Common Lisp Music (Schottstaedt, 1991).

MUSIC EDITING

Since, unlike Common Music, musical structure in Stella "survives" the composition of musical output it is available for editing operations as well as output realization. All structures in Stella may be archived and then reloaded at a later time, in much the same manner that a document may be tried and reloaded into an editor. This archived file is an ascii file containing a Lisp program that, when reloaded (evaluated), recreates the structures that created it. The archive file can be edited with any text editor and compiled to a Lisp binary file for more efficient loading. In order for objects to be edited, they must first be referenced in some manner. An Ed-Mixin class is provided that allows any inheriting to associate a unique
The basic distinction in Stella's representational scheme is that made between classes of objects which in some set reflect basic compositional data and classes of objects that implement collections, or groupings, of other objects. This basic distinction is represented by two classes in Stella: Element and Collection. An Element represents an "atomic" (so Stella) structure that cannot contain other Stella objects. This does not mean that an Element does not contain further "subdivisions" -- for example, into slots and values or parameterized data for instruments -- but only that Stella's mapping, editing and selection functions operate on this object in a basic structural unit. In contrast to an Element, a Collection represents a grouping or aggregate of zero or more sub-objects. Sub-objects in a Collection may themselves be Elements or other Collections. A Collection may serve simply to provide a convenient editing metaphor; working with a collection is easier than working with its objects individually. More importantly, Collection also serve as a root class for representing aggregate components such as vessels, mixes, sections, etc. Though a Collection class may contain further Collections and other objects of classes does not determine how the elements are actually collected or contained or in what manner they are accessed, or selected. These distinctions are left up to subclasses of Collection to implement. There are currently three basic subclasses, Thread, Menu and Composition. It is likely that there will be much more to be done with the Collection class. A Collection is conceptually any type of item stream pattern type in Common Music. Collections are really just subclasses of Common Music's Item Stream that have additional methods defined for archiving and loading, and that have the specialized capability of managing not only item selection that cannot contain other Collections, but also other Collections and other items that are not of a streaming type. Common Music's Collections have a layer of item stream pattern type: Collections on the functional pattern type, and so on the item stream accessing function, ITEM, will work for collections as well as for the standard classes of item stream, except that the item returned will be an Element with an unbonded Time attribute. Selecting the next Element in score processing is really no different than selecting the next item from an Item.
The following is an abbreviated diagram showing some of the most basic class relationships in Stella, followed by a brief description of the more important subclasses of Element and Collection:

```
Stella-Object
   \\
Collection       Element
   \\
Thread       Computation       Merge
   \\
Note       Default       Rest
   \\
File       Generator       Score
```

Note

The Note class is the root class for specializations of musical representations that are meant to be output to some program like the MusicKt, CML, Csound or MIDI for sound synthesis. Notes have no notion of absolute time, but only of time increment. Since Notes maintain only a time delta, they may be copied/duplicated to different locations in the composition without having to recompute absolute time values, and may be structure-shared across different containers.

Default

A Default represents musical data that is to be merged with other (possibly incomplete) specifications to produce a fully specified output event. If at selection time a Default is currently in effect, Notes of compatible class will merge any unbound attribute (an attribute which the composer has not explicitly set) with the current default's value for that attribute in order to produce the fully specified set of attributes. The value merging is nondestructive. Defaults are similar in spirit to "initial values" for structures and class instances, except that initial values hold at object creation time, and defaults hold at score evaluation time. Defaults permit default runtime parameter values to be set and are useful for exploring global effects of changes in data without modifying the actual attributes of Elements.

Thread

A Thread is the simplest type of Collection; it simply arranges and accesses its sub-objects in sequential order. When a Thread is requested to produce an Element it evaluates its sub-objects (which may involve recursion) to return the next Element. If an Element is found, its time is set to the Thread's current time plus offset, and the Thread (non-increment) its local time by the rhythmic value of the selected Element, if any. When a Thread reaches the end of its material, or if it cannot return an Element, it deinitializes itself and signals its caller that it has no more Elements to return until it is reinitialized. When an uninitialized Thread is called upon to produce an Element, the Thread caches the current time as its time offset, and the process continues.

Merge

A Merge is a Collection that processes its sub-objects in scheduling queue. Merges schedule their objects such that inferior Threads or Generators are run at the proper (local) time and that a single time ordered stream of Elements results. This merging is "virtual" in the sense that the resulting stream of events is not precomputed or cached as new structure, the Merge itself represents this cache. Of course it is possible to run a Merge and cache its resulting output events in a new Thread, which would then reflect one "version" of the Merge's output. All of the sub-objects in a Merge refer to local time kept by the queue, but output events returned by the Merge are incremented by the queue's current time offset to reflect its current time in the realization process. Scores are Merges with a few extra bookkeeping attributes. Merges may be elements inside other containers.

Generator

A generator is a Thread and a Computation that, if it has no Elements, or its current Elements have been marked as "forgotten", will reinitialize itself to run and perform some user specified computation (the returns data each time a new Element is requested). The generator then creates an Element of the appropriate class for this data, adds it to its current element cache, and returns the Element to the caller. If the user's Computation is invoked but signals that it has no more data to generate, the Generator deinitializes itself, signals to its caller that it has no more Elements to return, and the current contents of the cache are marked as invalid. Subsequent calls to the Generator will simply read Elements from the cache as a basic Thread, until such time as the Generator is told to forget its current cache. Generators may therefore be viewed as Computations with a history. This is particularly useful in situations where a composer wishes to employ random selection in the exploration of musical ideas: algorithms employing random selection can be frozen if the current results were particularly satisfactory, or forgotten if they were not. If the composer wishes to keep the current contents as a possible version but free the generator another computational pass, then either its Elements may be copied to a new Thread or the generator itself...
THE TOP LEVEL INTERPRETER

Though Stella is not an editor, a command interpreter has been implemented for top-level interaction with the objects in the system. This interpreter runs at the Lisp "top-level" and allows both standard Lisp expressions and Stella commands to be evaluated. The command interpreter is designed to run in any Common Lisp, and provides a requestable working environment for hardware and Lisp combinations that will not support the upcoming graphics editor I plan to implement later this year. The interpreter's commands are classified in five broad categories according to usage: information commands, navigation commands, editing commands, output commands, and archiving commands. The top-level interpreter acts as an inspector and structuring editor element: it displays, expanded, created, modified, performed, archived using a simple command syntax. One advantage the top-level interpreter is this, except for simple editing expressions, Common Lisp's "problematic" Lisp syntax is avoided, and a non-programming computer can perform complex editing operations without knowing any Lisp programming. All commands share the basic syntax: Command (reference arguments, where nReferences is the set of elements on which to perform Command and arguments. Conventions are implicit in mapping operations. A reference may be specified as a name, or a relative position. If no reference is supplied, the current focus element is used, otherwise the references are merged with the current focus container to obtain the set of elements to operate on. For example:

> Show
display information about the current focus object and
> Show 1/4
> rise(l:3,10,1)
> would display the first 4 objects in the current focus container and the first three, tenth and last objects in the container named "the". Similarly,

> Transpose risea -6
would shift all elements in the focus that are capable of responding to the generic transpose function 6 degrees lower in the current scale object, and

> Sec * dur (random 1.0) note (pitch c3) (gs b in heap)
would set all the current container's elements with a dur attribute to a random amount less than 1.0, and note attributes to pitches read from an item stream in the heap pattern type. One possibility which the top-level interpreter presents for Stella that I will explore in the future is, since Stella can respond to simple input strings, she could easily interact in multi-processing environments as a compositional "listener" that responds to command strings and from other processes or other command interpreters.

RELATED WORK

Since Stella occupies the wide spectrum between algorithmic composition and even-list editors, there are large numbers of programs, languages, and environments that demonstrate similar functionality and share similar representation metaphors. Though PLA (Schoosmaeder, 1985) began CommonMusic and hence item systems, and Stella is an extension of this work, PLA and its notion of streams is probably the intellectual ancestor of Stella, but it is no closer the clearest match to the current system's overall functionality and spirit of design. There are a host of current software systems that utilize the hierarchical notion of abstraction, subtraction, and element, indeed, it is hard to think of how this could not be so given its explicit embodiment in music performance and notation. I would cite David Jaffe's Music Kit as a particularly rigours and wide-spread implementation of the hierarchical metaphor in music software. The Music Kit uses hierarchical representations in its compositional classes: Score, Part, Phrase, Note, Parameter, as well as in its performance representations: Orchestra, Conductor, Performer, Instrument, Synthesizer. However, the Music Kit is not designed for interactive editing, nor as a general compositional "environment". The system that most closely matches CommonMusic/Stella in functionality if not in representation scenario is Stephen Pope’s MUSIQUE (Pope, 1981). Like CommonMusic, MODE is an open-ended, object oriented, synthesis-independent composition system that represents a range of musical structures from simple "event lists" to algorithmic compositional units. Threads are equivalent to EventLists in MUSIQUE, and MUSIQUE is simpler, I believe, across between ParallelEventList and EventSubtractor. Generator's in Stella are similar to EventGenerators in MODES although it is unclear to me if EventGenerators use a lazy evaluation scheme or not. MODE also supports editing operations, and can archive all its representations for later re-hushing. There are several Lisp programs that also share similarities to Stella Flavors Bunch (Fry, 1984) is a music composition system implemented in Symbolic's Flavor system, that has hierarchical representations, processes, and supports editing operations as well. Flavors Bunch also has the notion of default musical material, but its defaults are instance attributes, not "update" information at score realization time. FORMES (Wobber, Conejo, 1984) is also a object oriented Composition system that performs scheduling and implement hierarchical structures also found in Stella, as well as sequential and parallel accessing.
FUTURE DIRECTIONS

Stella is still in the first phase of development, but the essential formalization has been implemented along with most of the basic editing operations. Further work will be done on the implementation of different patterns of collections, file tracking, thread recording, realtime connections to the MusicKit and MIDI, and linkage with the Common Music score file generation system such that Stella objects can be used by scores and parts in the older system. The most significant development effort will be designing a graphic editor for Stella. This work will most likely be implemented on the SGI Indigo.

REFERENCES


