Abstract

The TTree is an active data structure devised by an hierarchical organization of musical objects as computer music scores. It achieves exceptional flexibility and generality by making all assumptions whatever about the nature of those objects and, literally, 'space-time'. As such, the structure is capable of utilizing the score as its own data base.

In the present report, TTree is introduced through the development of a simple compositional example. A formal definition of the structure in the context of an mathematical theory of language is presented. Finally, two related TTree-based compositional mapping devices are illustrated as evidence of that versatility.

Background

The TTree concept arises from a view of computer music systems as works of art in their own right, independent of whatever their technological merit might be. Designing such systems can be an exceedingly challenging if not confounding task. A contemporary manifestation of the impossibility of the ideal is the existence of hierarchically organized, they can be judged with such cause for the logical idea that reflects for the music which they essentially produce. TTree, in particular, embodies a musical composition/synthesis system some of the resistance to the hierarchically modeled musical form so prevalent in contemporary music theory.

TTree: An Informal Introduction

TTree is a strictly binary tree built of two different node types. Events are the tree's terminal nodes; they represent audible sounds, with their initial "T" from "Sound" or "Event". Events are internal nodes. Through their hierarchically organized coordinates, they structurally define musical objects by determining their starting times, hence their initial "T" from "Temporal" or "Time". Events maintain 4 state-variables:

* their name
* their time value, called "wait"
* a pointer to another TEvent, called "then"
* a pointer to either a TEvent or aEvent, called "up"

TEvents, like components of aTTree, can be depicted graphically by writing their entire together with arcs representing their two-pointers: a horizontal arc to the right for "then" (labelled by the event's "wait" value), and a downward vertical one for "up". The following figure shows a standard rooted binary tree form as a TTree.

Figure

```
  expansion  \---development---\---corpus---\---ends
```

The structure of this figure can be read directly from the tree. "Figure" is expansion wait 20 then develop- ment wait 2 then event 10 then ends (note that the "and" and "wait" variables of TEvents figure and "ends" are analogous). Now let us focus on the "event" TEvent, making it the root of a new subtree:

```
  \---stretto---
```

Stretto

In other words, "stretto is an event" wait 10 then \---stretto---". Now, for the 2-part stretto to actually be a stretto, the root entry must overlap the root. This is no problem for the TTree representation, however, provided that whatever "second" event turns out to be, it lasts for more than 5 time units. The crucial point here is that any TEvent, its length value, is determined relative to its duration. Rather, it specifies a non-negative time delay until the beginning of the TEvent designated by its "then" pointer.

TEvents do not specify duration. For all we know, the root entry could last anywhere from microseconds to millennia. So far, the TTree figure is pure structure—we have said nothing about what it actually sounds like!

This is the job of SEvent. As SEvent is any hierarchically organized system, it makes sense for a particular sound synthesis technology. Emphatically, this definition carries no implications whatever as to what it is a musical note, at least not in the conventional sense. Nevertheless, to keep this docu-
since in the realms of the familiar, let us imagine that SEvents are notes, just as they used to be: discrete musical events having definite pitch and duration. Graphically, such SEvents can be represented as (pitch, duration) pairs. To illustrate, let us complete the strobe scheme by composing "alloEntry" and "trueEntry" as follows:

\[ \text{seventy} \]
\[ \text{alloEntry} \rightarrow \text{trueEntry} \]
\[ \text{C} \rightarrow \text{G,3} \]

Since "alloEntry" and "trueEntry" overlap by two time units, this is a true strobe. Though admittedly no conceptual masterpiece, the scheme clearly illustrates the difference between a TE vent's "with" value and the duration of an SEvent, which is the SEvent's own private property. The distinction is crucial to the feasibility of the TTree concept. SEvents can be anything we wish: a call to a random procedure, an instruction to play a sound file from disk, a message to a robot conductor instructing it to cue in the first violin. TTrees provide the structural and temporal organization, SEvents fill the content.

Two other unique properties of TTrees are worth pointing out. First, note that the TTree is simply a binary tree 'in hiding': it has been rotated counter clockwise a quarter-turn from the way such trees are normally presented. This rotation is important, and is no accident—it preserves the conceptual habits of vertical simultaneity and horizontal succession so deeply ingrained in musicologists habituated to common music notation. Second, the TTree differs from most time-structured musical scores by placing time along the branches of the tree rather than between them. To illustrate, consider the following musically-relevant tree structure:

```
  figure
    \[ \text{expos} \rightarrow \text{dev...} \rightarrow \text{strobes} \rightarrow \text{code} \]
  \[ \text{alloEntry} \rightarrow \text{trueEntry} \]
  \[ \text{G} \rightarrow \text{C} \]
```

With the TTree, we decide when an event begins in relation to any other event by simply adding up the "with" values along the path between the two events (remembering that vertical, or "strokes" are not defined at all). The pitch C of the final strobe, for example, starts 30 + 5 = 35 time units after the beginning of the figure. No such simple formula exists for the multi-hopping tree. Although both trees can claim all the advantages that hierarchically structured trees have over strictly linear ones, only with the TTree is this complexity made perfectly explicit, resulting in a structure which is at once more intuitive and more computationally efficient than its multi-branding counterpart.

T-Trees: A Formal Definition

In the section, T-Tress are defined in the context of normal language theory. The reader unfamiliar with the theory may wisely skip this material without loss of continuity. The formalism, adopted here is taken from (Rigdon and Ulmian, 1973).

The idea of describing a data structure for computer music is the form of a grammar was first proposed in (Brosen et al., 1977). A T Tree grammar Q is a restricted context-free grammar of the form:

\[ Q = \langle \Sigma, E, S, \Sigma^* \rangle \]

where:

- \( \Sigma \) is a set of SEvents,
- \( E \) is a set of SEvent symbols,
- \( \Sigma^* \) is a set of productions of the form
  \[ A \rightarrow \tau, A \in E, \tau \in \Sigma \] .

\( S \in \Sigma \) is \( \Sigma^* \)'s grammar's start symbol.

SEvents are the grammar's non-terminal symbols—SEvents are the terminals. Productions in the grammar are restricted to a special normal form. Similar to Chomsky normal, the right-hand side of all productions in a T grammar must contain either a single terminal SEvent or an arbitrary number of non-terminals SEvents (Chomsky normal), no more than 2 non-terminals may appear on the right-hand side of a production).

---

185

ICMC Proceedings 1988
A TTree is a tree corresponding to a derivation in a T grammar. This correspondence, however, is quite different from that which exists between a derivation and a tree as conventionally understood in formal language theory. TTrees are strictly binary, and vertical and horizontal arcs replace the usual left-child and right-child arcs of most binary tree representations. The application of a T grammar production of the form $A ightarrow T_{1}$ will have a vertical arc from $A$ to $T_1$, a horizontal arc to $T_2$, $T_3$, a horizontal arc to $T_4$, and so on. Thus: 

$$A \rightarrow T_1 \rightarrow T_2 \rightarrow \ldots$$

The primary purpose of TBoxes is to situate SEvents in time. To this end, a syntactic-negative 'wait' variable is associated with every TEvent. The time of any SEvent $e$ is the sum of the values of all 'wait' variables associated with those TEvents having horizontal branches or the path from $e$ (the static symbol) to $e$.

Implementations

1. AMA

A Music Automata (AMA) is a TTree-based compositional environment for the Synclavier II synthesizer in the electronic music studio of McGill University's Faculty of Music (Detert, 1985). The system provides the composer with a set of special-purpose editors for creating and modifying TEvents and SEvents. Once a set of these objects is defined, they are assembled into TBoxes using a command language which omits the production rules of context-free grammars. These commands effectively present the TTree contents from a datastructure in a language. To illustrate, one figure example is created by typing the following lines to the AMA system:

```
figure 1 - music development system code
struct amake_tempEntry
struct Entry
entry Entry = G
```

A graphic representation of the TTree is shown on the computer screen as these commands are typed. Acoustic feedback is immediate—either the entire tree or any of its branches can be performed in real time at any point in the process.

In AMA, SEvents take the form of recipes for the Synclavier II's digital oscillators. These oscillators have a hardware configuration which AMA models directly—these are preserved in the SEvents' structure for each of the oscillators' input registers. AMA, then, rather than abstracting away these characteristics, elevates them to instruments in their own right, offering the computer direct control over every aspect of their functionality.

2. Forrest

TTrees borrow heavily not only from formal language theory but from object-oriented programming principles as well. In consequence, a research project in implementing them in an environment specifically supporting the object-oriented paradigm was initiated. The case with which musical objects can be represented in the language Smalltalk-80 (Goldberg and Robeson, 1982) had already been clearly demonstrated, most recently at (Pope, 1987) and (Scalzi, 1987). For this reason, a set of Smalltalk-80 classes dubbed 'Forest', was implemented at Stanford University's Center for Computer Research in Music and Acoustics.

ICMC Proceedings 1983
Plate II. Three of Forest's SEvent editors.

Plate II shows three of Forest's built-in editors for manipulating SEvents. The window labeled 'collection' is a code SEvent, whereas a note-list SEvent, while 'tick' is a common music notation SEvent which uses proportional notation on a grid and spanning the entire MIDI pitch range.

In Forest, the 'wait' field of SEvents are themselves holds on objects capable of responding true or false to the message 'if'. Whenever a Event receives the tick message, it immediately replies the message to its own 'if' object. The relationship to its 'then' object, however, if its 'wait' object responds to tick with false. The initiation method for the tick message in class TEvent, then, is simply:

tick

TEvent performance is accomplished by repeatedly sending the tick message to the root TEvent. The initiate method for tick, asked by its 'wait' object, provides explicit control of the propagation of this message through the tree. The most common form of 'wait' objects, called a 'timer', responds to 'tick' by decrementing an internal counter, assessing true if the counter is greater than 0, else false. Other kinds of behavior can be easily defined by the user—

Forest provides a simple mechanism for coding new 'wait' objects. With the full expressive power of Smalltalk-80 at their disposal, such objects are capable of testing input from keyboard, mouse, or any
other device, allowing the embedding of real-time control into the high-level structure of the composition.

As described in (Bristor et al., 1977), tree-structured score representation schemes, with their implicit inheritance capabilities, are capable of organizing much more than just the temporal structure of music. In Furtet, inheritance occurs as a by-product of a simple yet powerful inter-object communication scheme. Every Forest object, whether SEvent or TEvent, includes an instance variable called 'ancestor' which points backward to that event's parent. This variable, together with the 's' and 'then' variables of TEvent, allows any event in a TTrees to communicate with any other event. To illustrate, referring to Figure 1, the SEvmt named 'collage' can communicate with the exposition's 'theme' by following the ancestor links backwards through development to exposition, then following the 's' link down to 'group1', the 'Then' link to 'group2', and finally the 's' link to 'theme'. The whole path can be described in Smalltalk as 'ancestor ancestor is then s'.

Every Forest object maintains a dictionary, called the 'legacy', in which user-defined blocks of code can be stored, indexed by symbolic name. To illustrate, common music notation SEvets, as part of their response to the 'tick' message, will evaluate the code associated with the symbolic name 'compose' in the legacy dictionaries of SEvets along the chain of ancestors leading back to the tree's root. By entering code into the appropriate legacy dictionary, the composer can effect transposition of any desired section of the composition. And because this inheritance mechanism is completely general and extensible, such transposition but any transformation of the composer wishes to define can be quickly and easily implemented within Forest's framework.

Acknowledgements
The author gratefully acknowledges the Social Sciences and Humanities Research Council of Canada for their support of his ongoing research as Stanford. Thanks to Bill CCRMA for all their help.

References