MOOD: A Concurrent C++-Based Music Language

David Anderson
<anderson@icsi.berkeley.edu>
Sonic Solutions
1891 E. Franciso Blvd.
San Rafael, CA 94901

Jeff Bilmes
<bilmes@amt.mit.edu>
MIT Media Lab
Massachusetts Institute of Technology
Cambridge, MA 02139

Introduction

MOOD (Musical Object-Oriented Dialect) is a C++ class library for computer music [1]. It runs on SPARC, MIPS, and MC680x0-based UNIX machines and on the Apple Macintosh, and uses MIDI I/O. It is designed for algorithmic composition, interactive systems, and cognition research, and is well-suited to any application that needs concurrency and precise timing control. MOOD borrows many ideas from FORMULA [2].

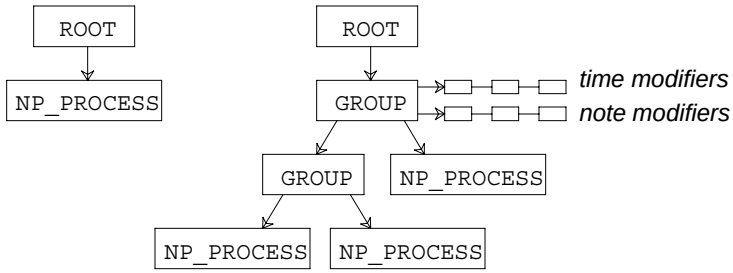
We have interfaced MOOD to Tcl (Tool Command Language), an embedded interpreted command language [3], producing a shell that allows you to interactively start and stop MOOD processes by typing Tcl commands. You can also define ''scripts'' that perform sequences of musical actions.

Process Scheduling

MOOD provides lightweight processes sharing a single address space. These processes are preemptively scheduled (one can interrupt another at any point) according to a real-time policy. Each process is represented by a C++ object. There are several types of processes, with corresponding C++ classes. Keyboard and MIDI input is handled by *real-time processes* (class `RT_PROCESS`). *Note-playing processes* (class `NP_PROCESS`) generate streams of notes (`NOTE` objects).

Hierarchical Structure

`NP_PROCESS`es can be collected into hierarchical group structures, with associated virtual time systems and nested musical transformations. Many such trees can exist at once. The internal nodes of a tree are `GROUP` objects, and its root is a `ROOT` object:



The simplest configuration is a single `ROOT` and `NP_PROCESS` (shown above, left). You can attach *modifier* objects to `GROUP`s and `NP_PROCESS`es. There are two types of modifiers: *time modifiers* operate on time intervals (changing tempo or note timing), while *note modifiers* operate on notes (changing their volume, pitch *etc.*). Modifiers may themselves be processes (see below). Modifiers are applied going up the tree. Thus the time values (lengths of notes and rests) generated by a `NP_PROCESS` pass through a pipeline of modifiers before emerging as real-time intervals, while the notes themselves pass through another series of modifiers before being played via MIDI.

Time and note modifiers are derived from a base class `MODIFIER`. Subclasses have been defined to control rhythm, volume, articulation, time shifting, and tempo. Some of these classes are simple objects (*e.g.*, adding a constant to volume); others use a separate process to generate the parameter control. It is often handy to combine several note modifiers. The class `MULTI_MOD` (derived from `MODIFIER`) represents a set of `MODIFIER`s that are applied in sequence.

Playing Music

Note-playing processes play notes using a notation similar to C++ stream I/O [4]. `NP` is a predefined object with overloaded `<` and `<=` operators. `<` plays a note and advances (in time) by its duration; `<=` plays a note and remains at the same time. Compound musical units (chords, sequences, *etc.*) are represented by `PHRASE` objects, which can be played using the same stream notation. Pitches may be specified by integers (their MIDI note numbers) or by `PITCH` objects; Constant `PITCH` objects are predefined for pitch naming: `C5` is middle C, `+C5` is C sharp, `-E5` is E flat, *etc.* `RS` is a special pitch number indicating a rest. For example, this code plays a C major arpeggio, then a chord.

```
NP < C5 < E5 < G5;
NP <= C5 <= E5 <= G5 < C6;
```

Some C++ operators can be applied to `PITCH` objects. Each `PITCH` uses a `MODE` object to determine the meaning of these operations. For example, `p++` increases `p` by one mode step, `p+i` returns a pitch that is `i` mode steps above (below) `p`, and `p[i]` returns a pitch that is i octaves above `p`.

Rhythm-generating note modifier processes also use a notation modeled after C++ stream I/O. Operator overloading is used to provide a concise syntax, as shown below.

Example

The following MOOD program plays a whole note C, two half-note C#s, three whole-note triplets on D, and so on.

```
void rhythm() {         // rhythm-generating process
  for (int i=1; i<=16; i++)
     RP < w_/i%i;       // play i i-th notes
}

void notes() {          // note-playing process
  PITCH p = C5;
  for (int i=0; i<16; i++) {
    for (int j=0; j<=i; j++)
      NP < p;
    p++;                // go up a half-step for each rhythm
  }
}

main() {
    SCHEDULER::run(
        new ROOT(
            new NP_PROCESS((PROCEDURE)&notes, no_args,
                new RH_PROCESS((PROCEDURE)&rhythm, no_args)
            )
        )
    );
}
```

The main program creates a `NP_PROCESS` that executes the function `notes()`. It has an attached note modifier of class `RH_PROCESS`, which executes the process `rhythm`.

References

[1] D. P. Anderson and J. Bilmes, ‘‘Concurrent Real-Time Music in C++’’, *USENIX C++ Workshop*, June 1991.

[2] D. P. Anderson and R. J. Kuivila, ‘‘FORMULA: a Programming Language for Expressive Computer Music’’, *IEEE Computer*, June 1991.

[3] J. Ousterhout, ‘‘Tcl: An Embeddable Command Language’’, *Proceedings of the 1990 Winter USENIX Conference*, Washington, DC, , 133-146.

[4] B. Stroustrup, *The C++ Programming Language*, Addison-Wesley, 1986.