Concurrent Real-Time Music in C++

David P. Anderson †‡ *Jeff Bilmes* †
<anderson@snow.berkeley.edu> <bilmes@icsi.berkeley.edu>

†International Computer Science Institute
1947 Center Street
Berkeley, CA 94704

‡Computer Science Division, EECS Department
University of California at Berkeley
Berkeley, CA 94720

Abstract

MOOD is a C++-based programming system for algorithmic and interactive music generation. MOOD uses multiple concurrent processes to generate different aspects of musical structure (pitches, rhythm, dynamic variation, etc.). It is composed of three layers. Layer one supplies deadline-scheduled lightweight processes and real-time event generation. Layer two allows processes to be collected into hierarchical group structures, with associated "virtual time systems" and nested musical transformations. Layer three provides pitches, scales, notes, rhythm specification, and higher-level musical abstractions. MOOD derives several benefits from C++ features such as inheritance and operator overloading: 1) a simple and versatile syntax for music representation; 2) a clean, layered structure for the internal scheduling mechanisms; 3) easy factorization of the machine-dependent parts (MOOD now runs on Sun 3 and 4 workstations under UNIX, and on the Macintosh).

1 Introduction

MOOD (Musical Object-Oriented Dialect) is a programming system for note-level computer music (e.g., computer control of MIDI [Int89] synthesizers). Unlike standard music sequencer programs that represent music as lists of note data structures, in MOOD the music is represented by the code itself; hence MOOD can specify musical algorithms as well as scores. We intend MOOD to support a variety of musical activities, including 1) algorithmic composition; 2) interactive performance environments, and 3) programmed score interpretation.

We implemented MOOD as a set of C++ [ES90] classes. As discussed by Pope [Pop89], object-oriented programming languages are useful for computer music since they provide design techniques such as composition, refinement, factorization, and abstraction. Our additional reasons for using C++ include:

1) **Familiarity:** Programmers familiar with C++ can immediately use MOOD. Furthermore, MOOD can benefit from C++ development activities in other areas, such as user-interface toolkits.

2) **Clean syntax:** The operator-overloading features of C++ allow us to provide concise and intuitive syntax for common musical structures.

3) Speed: Optimizing compilers are available for C++, and there is no built in garbage collection. These properties improve timing accuracy.

4) Portability: C++ runs on a variety of machines and operating systems, and makes it easy to encapsulate system dependencies.

5) Extensibility: Inheritance make it possible for programmers to extend and customize the features of MOOD for particular musical styles or applications.

MOOD is composed of three layers. Layer one supplies deadline-scheduled lightweight processes and real-time event generation. Layer two supplies hierarchical "virtual time systems" that allow the nesting of musical transformations. Layer three provides pitches, notes, scales, rhythms, and higher-level musical abstractions. Figure 1 shows the MOOD class hierarchy and its division into layers.

2 Layer One: Basics

Layer one provides the rest of MOOD with lightweight processes, accurately-timed event performance, and other low-level features. MOOD uses the scheduling model of FORMULA [AK90].

2.1 Real-Time Processes

MOOD uses several specialized types of processes. The `PROCESS` class abstracts the features common to these types. These include a stack, an SP save slot, and member functions for context switching and stack initialization. The `RT_PROCESS` class also inherits `SCHED_REQ`, adding the necessary state for real-time scheduling (see below). An `ARGS` object stores the arguments (and their number and types) to be passed to the initial procedure executed by the process. A process might be created as follows:

```
ARGS args;
PROCESS *p;
args << 5 << 3.5;
// the new process will execute foo(5, 3.5)
p = new RT_PROCESS(foo, args);
```

2.2 Process Scheduling

Real-time scheduling is encapsulated in the classes `SCHEDULER` and `SCHED_REQ`. `SCHED_REQ` abstracts the notion of "schedulable entity", with virtual member functions `run()` and `preempt()`. For example, the implementation of `run()` in `RT_PROCESS` simply switches to the process. A `SCHED_REQ` also includes `TIME` members `deadline`, `time_position`, `maxdel`, and `mindel`. `time_position` is the real time for which events (such as notes) are currently being computed by the entity. `time_position` may be greater than the current real time but if it exceeds it by more than `maxdel` (i.e. $time\_position > currentSVT + maxdel$), the entity is temporarily put to sleep. `deadline` is the entity's scheduling priority, and is equal to `time_position` - `mindel`; thus `mindel` can be used to prioritize processes (such as input handlers) with similar time positions.

Figure 1: The MOOD class hierarchy and layers.

Earliest-deadline-first CPU scheduling is used: a `SCHED_REQ` object runs only when it has the earliest deadline, and runs until it changes its time position and deadline using:[1]

```
    SCHED_REQ::set_time_position(TIME);
```

After an `RT_PROCESS` is created and initialized, it can be made runnable using:

```
    SCHEDULER::make_runnable(SCHED_REQ*);
```

Scheduling is *preemptive*: if a process with an earlier deadline than the current process P is made runnable (by an interrupt handler or by P itself) it preempts P.

Real time is expressed in units of system virtual time (SVT). The number of units of SVT per clock period can be varied slightly to phase-lock MOOD to an external timing source. The classes `TIME` and `TEMPO` represent times (absolute or relative) and time scaling factors respectively. On the MC68020, `TIME` and `TEMPO` objects are 64 and 32 bit fixed point integers; on the SPARC, they are 64 and 32 bit floating point values. C++'s operator overloading facilities make the implementations of `TIME` and `TEMPO` transparent to users of the classes.

When possible, events are computed before they are performed (i.e., the deadline of the currently running process may exceed the current SVT). The member variable `maxdel` determines the maximum amount by which an `RT_PROCESS` may run ahead of real time, and thus limits its response latency for asynchronous I/O events. `system_mindel`, a `TIME` member of `SCHEDULER`, is used to keep processes from falling too far behind schedule. If the following condition holds:

$$earliest\ deadline \leq currentSVT - system\_mindel$$

then SVT is not advanced.

2.3 Event Generation

To improve timing accuracy, MOOD separates the computation of a note and its parameters (pitch, volume, etc.) from the action that causes the note to sound. This is especially useful when using MOOD with a DSP system as a synthesizer. The class `TIMER` encapsulates timed performance of output actions, via

```
    TIMER::insert_request(TIMER_REQ*).
```

`TIMER_REQ` is an abstract class; its derived classes define "action routines" and their parameters. A Process plays notes at a specific time by: 1) advancing its time position, 2) computing the notes to be performed at that time, and 3) calling `TIMER` to schedule the playing of the notes. Hence a process is usually computing notes that will be sounded at a later time.

`TIMER` uses a separate *timer process* for event performance. On each clock interrupt, `TIMER` checks if any events are pending and if so makes the timer process runnable. The timer process checks if SVT has reached or passed an event's time, calling the appropriate routine if it has. The accuracy of event timing is limited by the clock period.

[1]We use the scope resolution operator here to show which class contains the member function.

2.4 Asynchronous I/O

MOOD provides asynchronous I/O, i.e., a facility for a process to do a blocking read from a descriptor without causing the whole UNIX process to block. This is encapsulated in the `IO` class, whose constructor takes a file descriptor and which provides `read()` and `write()` operations, implemented using `SIGIO` and `select()`. A flag indicates whether `read()` and `write()` operations should return when 1) some I/O has been done or 2) the entire request has been done.

2.5 UNIX Implementation

In the UNIX implementation of MOOD, the timer and I/O interrupts are signals. The signal handlers may wake up processes (e.g., the timer process). If one of these processes has a deadline before the current process, preemption is needed. This is done by changing `sc_pc` in the handler's `struct sigcontext` to the address of an assembly routine `preempt`. This routine saves the complete context of the interrupted process and does a (non-preemptive) context switch to the process with the earliest deadline.

MOOD uses *virtual interrupt masking* for critical sections. This technique uses *mask level* and *request* variables. If a signal handler finds that the mask level is nonzero, it sets a bit in the request word and returns. `mask_ints()` increments the mask level. `unmask_ints()` decrements the mask level and, if it is zero and the request word is nonzero, calls routines that do the work of the clock or I/O signal handlers.

UNIX is not a real time operating system, and there are noticeable timing delays when there is other system activity. On the SPARCStation, we have adapted MOOD to use a MIDI device driver that keeps an output queue in the kernel, and performs events at the hardware interrupt level. This greatly increases the accuracy and resolution of event timing.

3 Layer Two: Virtual Time

Layer one allows processes to schedule events in real time. Layer two extends this by supporting *virtual time systems*: coordinate systems for time that can run faster or slower than real time, modeling the way a musician varies tempo during a performance. Each virtual time system is represented by an object of class `VTS`, whose state includes *inner* and *outer* time positions (generalizations of layer one's deadlines). An object of class `FUNCTION` defines the mapping (when inner time is incremented by x, the `FUNCTION` "deforms" x and the result is added to the outer time).

`FUNCTION` is an abstract class. Simple derived classes multiply x by a constant, or call a function to deform x. `TD_PROCESS` deforms x by doing a coroutine switch to a *time deformation process*. This process defines a "tempo function" by calling primitives

```
    // linear tempo change
    td_seg(TIME dt, TEMPO start, TEMPO end);
    // pause
    td_pause(TIME dt);
```

whose implementations handle the context-switching details. Finally, `COMPOUND_FUNCTION` encapsulates a list of `FUNCTION` objects, which are called in order when the `COMPOUND_FUNCTION` object is called.

Figure 2: Examples of a single virtual-time process (a) and a group of three such processes (b). Attached to each VTS are two FUNCTIONs: a "time deformation" that controls its tempo, and a "modifier" that is applied to any notes played by descendant processes.

Processes can be organized into a hierarchy of groups, each with their own virtual time system (see Figure 2). A top-level group is represented to the SCHEDULER as a VT_SCHED_REQ object, which inherits from SCHED_REQ. This VT_SCHED_REQ is linked to a VTS whose outer time corresponds to real time. Each VTS represents (and contains pointers to) either a group of VTSs or a single VT_PROCESS. A FUNCTION attached to a VTS affects all of its descendants.

A VT_PROCESS can schedule events using VT_PROCESS::timer_request(TIMER_REQ); This stores the (real-time) time position of the caller's topmost VTS ancestor in the TIMER_REQ, then calls TIMER::timer_request(). A VT_PROCESS can change its time position using VT_PROCESS::time_advance(TIME dt). Timing is expressed in the inner time of its VTS. The mapping to real time is the composition of all the FUNCTIONS along the branch of the VT_PROCESS.

When a VT_SCHED_REQ is executed, it does a context switch to its "earliest" VT_PROCESS descendant, whose time position determines the time position of the VT_SCHED_REQ and therefore its deadline. For efficiency, VTS groups are time-ordered lists, and each VTS stores a pointer to its earliest VT_PROCESS descendant.

Each VTS also includes a *note modifier* FUNCTION used by layer three (see below). Finally, layer two allows a VT_PROCESS to schedule *future actions* at times other than its current time position. These are stored in a "future action queue", which is traversed by VT_PROCESS::time_advance. This makes it convenient to schedule key-up commands whose timing may be intermixed with future key-down commands.

4 Layer Three: Music

Layer three of MOOD provides music-specific features. The design uses abstract classes and inheritance to provide an "open framework" in which new features can be added easily.

4.1 Note Playing Processes

`NP_PROCESS` (note-playing process) adds musical features to `VT_PROCESS`. A `NP_PROCESS` contains a `NOTE_MAKER` and a `NOTE_PLAYER` object as instance variables. Objects subclassed from `GENERATOR`, representing notes or groups of notes, are sent to a `NOTE_MAKER` using `NOTE_MAKER::operator<(GENERATOR&)` for successive events and `NOTE_MAKER::operator<=(GENERATOR&)` for simultaneous events. The `NOTE_MAKER` will then ask the `GENERATOR` to supply it with event information by calling the virtual member function `GENERATOR::apply(NOTE_MAKER&);`. Subclasses of `GENERATOR` redefine the `apply()` routine to send their component objects back to the `NOTE_MAKER`. Using this method, subclasses of `GENERATOR` may produce a variety of musical structures.

A `NOTE` object represents a single note. It adds duration, volume, and duty (the fraction of duration during which the note sounds) to `PITCH` objects. A `NOTE_MAKER` object has specific versions of `operator<()` for `PITCH` objects, subclasses of `NOTE` objects, and `int`s. These routines can take a `PITCH` or a partially constructed `NOTE` (e.g., lacking volume) as input, and produce a fully constructed `NOTE`. They do this in the standard version of `NOTE_MAKER` by applying the note modifier `FUNCTION`s of the ancestor `VTS`s, in order. For example, these `FUNCTION`s might modify the volume of notes, supply a duration, or change the duty.

A `NOTE_PLAYER` object takes complete `NOTES` as input and handles them in a subclass-dependent way. The standard `NOTE_PLAYER` class plays the note via MIDI, using `TIMER` to schedule the output actions (usually note on and off MIDI events) and `SCHEDULER` to advance its time by the duration of the note. Other subclasses write the note to a file, or send it to another process.

The `NP_PROCESS` typically computes a sequence of `PITCH`s and uses its `NOTE_MAKER` to convert these to `NOTE`s which then feeds them to its `NOTE_PLAYER`. Rhythmic figures are obtained by note modifier functions that use a `SG_PROCESS` (sequence generator process) set up by the `NP_PROCESS`. A `NP_PROCESS` may access its `NOTE_MAKER` using the symbol `NM`. Assuming `C`, `D`, `E`, and `F` are `PITCH` objects, a succession of notes may be played as:

```
    NM < C < D < E < F;
```

4.2 Pitches, Modes, and Temperament

`PITCH` objects, representing frequencies, maintain a pitch number which can be used as a MIDI note number or as a value to generate a frequency. `PITCH` objects also contain, as member variables, `MODE` and `TEMPERAMENT` objects. The pitch number of a `PITCH` object can change in either pitch steps (called pitch step deltas) or mode steps (called mode step deltas). When a `PITCH` changes by i pitch steps, the frequency the `PITCH` represents usually changes by i half-steps in the chromatic scale. When a `PITCH` object changes in mode steps, the pitch number must change by an appropriate number of pitch steps. A `MODE` object acts as a function with state that maps from mode step deltas to pitch step deltas. The state is simply the current position in the `MODE`. Therefore, when a `PITCH` object changes by n mode steps, it notifies its `MODE` object who adjusts itself by n mode positions and supplies a pitch step delta back to the PITCH object. Predefined global MODE objects include Ionian through Locrian, and, although they are not modes in the same sense, Chromatic, Major, Minor, HarmonicMinor, MelodicMinor, LydianMinor, Blues, Gypsy, Jewish, etc. The "..." mechanism is used in MODE constructors. The first argument gives the number of mode steps, and the following arguments supply the steps themselves. For example:

```
const MODE Chromatic(12,1,1,1,1,1,1,1,1,1,1,1,1);
const MODE Ionian(7,2,2,1,2,2,2,1);
const MODE HarmonicMinor(7,2,1,2,2,1,3,1);
const MODE Gypsy(7,1,3,1,2,1,3,1);
```

The TEMPERAMENT of a pitch may be used to generate micro-tonal and non-equal tempered music. When constructing PITCH objects with non-default TEMPERAMENTs, the initializer PITCH object can act as the pitch base. The pitch base is the frequency that corresponds to the first scale value the TEMPERAMENT defines. One may optionally provide an offset into the temperament that specifies where in the scale the constructed pitch should be placed. For example, if C and D are PITCH instances:

```
// p uses the C natural Major scale
PITCH p(C,PureMajor);

// q uses the D natural Major scale
PITCH q(D,PureMajor);

// r uses a Natural Major scale whose second frequency is C.
// Note: The frequency of one pitch step less than r is not the
// same as the frequency of B.
PITCH r(C,PureMajor,1);
```

Thus, the constructed PITCH object along with its TEMPERAMENT defines a mapping from integer pitch numbers to frequencies. When a PITCH is used with an FTEMPERAMENT, the FTEMPERAMENT may be used to calculate the frequency in Hertz of the PITCH. This will be useful for interfacing MOOD with a DSP system. Global FTEMPERAMENT objects are pre-defined including EqualTempered, PureMajor, PureMinor, and MeanTone. The "..." mechanism is used in FTEMPERAMENT constructors also. The first argument gives the units in divisions per octave (1200 corresponds to cents), the second argument gives the number of frequencies per cycle, and the following arguments supply the multiplicative factors for computing the successive frequencies. The following is an example of constructing FTEMPERAMENT objects.

```
const FTEMPERAMENT EqualTempered(1200,12,100,100,100,100,
    100,100,100,100,100,100,100,100);
const FTEMPERAMENT PureMajor(1200,12,70.673,133.237,111.731,
    70.673,111.731,70.673,133.237,70.673,111.731,133.237,70.673,
    111.731);
```

The default MODE for a PITCH is Chromatic, and the default TEMPERAMENT is NullTemperament (does nothing). These defaults are implemented using C++ default arguments in the PITCH constructors.

4.3 Musical Operators

Operations on musical structures applicable to a NOTE_MAKER are abstracted by the class OP_GENERATOR (operable generator). Using C++ operators, musical structures may be manipulated in novel ways.

4.4 Operations on PITCH objects

Abstracted by OP_GENERATOR, many C++ operators are defined on PITCH objects. For example, if p is a pitch, p++ increments the pitch by a mode step, p-- decreases it by a mode step, +p returns a sharpened pitch (a pitch that is one pitch step greater), -p returns a flatted pitch, p += i increases p by i mode steps, p -= i decreases it by i mode steps, p %= i increases in MODE steps but wraps at the octave, p <<= i increases p by i semitones, p >>= i decreases in semitones, p[i] is i octaves above p, and p[-i] is i octaves below. Assignment operators allow both PITCH and MODE objects to be assigned to PITCHs (assignment of a MODE to a PITCH only changes the MODE). Relational operators are defined as expected (< means less in pitch, etc.). Most binary operators are also defined as expected on pitches, preserving mathematical identities where possible. Therefore, pitch equations and expressions may be used. For example:

```
if ( p+3 >= A[i]-1 )  // i is an int
    NM < p+3 < p<<4;
else
    NM < p-2 < p>>3;
```

Global pitch constants (A through G) with default MODE and TEMPERAMENT are predefined. A[5] is 440Hz and C[5] is middle C. To reduce unnecessary bracket use, pitch constants C1 through B12 are also defined. Certain operations are not allowed on pitch constants. For example, the construct C++ is invalid since it is an attempt to modify the constant pitch C. C++ constant member functions make it easy to enforce this restriction.

With these constructs, we may write:

```
for (PITCH p = C[5]; p <= C[6]; p++)
    NM < p;
```

which will play a chromatic scale starting at middle C, and

```
PITCH p(Ionian);
for (p = C[5]; n <= C[6]; p++)
    NM < p;
```

which will play a C major scale.

4.5 SEQUENCEs, CHORDs, and MFILEs

SEQUENCE objects store an ordered sequence of OP_GENERATOR objects. A SEQUENCE will send its components to a NOTE_MAKER in the order they were loaded. Similar to a NOTE_MAKER, SEQUENCE objects are loaded with SEQUENCE::operator<(OP_GENERATOR&);. All of the operators defined by OP_GENERATOR may be applied to a SEQUENCE; the operation defined will affect all of the SEQUENCE's components. Therefore, one may easily manipulate melodies.

Figure 3: Chords of the C Harmonic Minor Scale

```
SEQUENCE s;
s < C4 < F4 < G4 < D4 < C5 < -B4 < F4;
NM < s;     // play the sequence
NM < s[1];  // play it an octave above.
NM < s<<1;  // play it a half-step down
```

Similar to `SEQUENCE` objects, `CHORD` objects cause its component's events to occur when applied to a `NOTE_MAKER`. A `CHORD` object's components play simultaneously however. Therefore, one may easily play the chords defined by a given mode (see figure 3).

```
CHORD ch;
ch <= C5 <= -E5 <= G5 <= B5;
ch = HarmonicMinor;
// Play the chords defined by the C Harmonic Minor scale
for (int i=0;i<8;i++)
    NM < ch++;
```

`CHORD` objects also define operators for changing the voicing and chord inversion. Subclasses of `CHORD` provide predefined chords. The root of these chords are given by a `PITCH` at construction time. For example, `MAJ7 maj7(C4);` produces a C Major 7th chord.

`CHORD_SEQUENCE` objects are used when both `operator<()` and `operator<=()` are needed to store a set of `OP_GENERATOR`s. `MFILE` objects can be used to store sequences or chords and play them at a later time. For example:

```
MFILE mf("myFile");
if (mf.open()) {
    PITCH p = C4;
    mf.truncate();
    while (p < C5)
      mf < p++;
    mf.close();
    NM < mf; // this will reopen and play the file.
}
```

`SEQUENCE`s, `CHORD`s, and `MFILE`s thus make it relatively easy to manipulate melodies, chord sequences, and pitches contained in files.

4.6 Rhythm

Note durations are either associated with `NOTE` objects when they are constructed, or are obtained by the `NOTE_MAKER` (via a modifier) from a `SG_PROCESS` (sequence generator process). The function executed by a `SG_PROCESS` specifies rhythmic figures using the two macros. `B(n,d)` specifies `n` note durations of length $1/d$ (e.g., `B(4,16)` specifies four sixteenth notes).

Figure 4: MOOD Score Example

`F(n,d)` specifies a n/d note duration (e.g., `F(1,1)` is a whole note duration, `F(3,8)` is a dotted eighth note duration, and `F(7,16)` is a double dotted quarter note).

5 Example

The following example demonstrates how MOOD may concisely represent musical figures. It shows how abstract musical structures may be defined (the definition of procedure `void playLine(PITCH&);`), and how concrete instances of the abstraction are made (creation of processes executing the procedure). Figure 4 shows the traditional notation.

```
// MOOD demonstration program.
#include "mood.h"

void seqGen() {
  B(56,16);  // generate 56 16th notes,
  B(1,2);    // and 1 half note
}

void playLine(PITCH& startPitch) {
  // set up an associated sequence generator
  AUX_INIT;
  SET_TSG(new SG_PROCESS((PROCEDURE)seqGen, no_args));
  SEQUENCE s1, s2;
  SEQUENCE bar1,bar2,bar4; // bar 1 and 3 are equivalent

  // load sequences with relative pitches
  s1 < startPitch < startPitch+1 < startPitch+2 < startPitch;
  s2 < startPitch < startPitch+2 < startPitch+1 < startPitch;
  // load bar 1 through 4.
  bar1 < s1++ < s1++ < s1; s1 -= 2; bar1 < s1++;
  bar2 < s1; s1 -= 2; bar2 < s1++ < s2++ < s2;
  bar4 < ++s1; s1 -= 2; bar4 < s1 < C4;

  // send sequences to our note maker
  NM < bar1 < bar2 < bar1 < bar4;
}

main() {
  NP_PROCESS *p;
  VT_SCHED_REQ *q;
  timer.start_clock();
  timer.set_tempo(2500000);
  ARGS args1, args2;

  // create the first process
  args1 < PITCH(C5,Ionian);
  p = new NP_PROCESS(std_note_maker, midi_out, (PROCEDURE) playLine, args1);
  scheduler.make_runnable(q);

  // create the second process
  args2 < PITCH(G5,Ionian);
  p = new NP_PROCESS(std_note_maker, midi_out, (PROCEDURE) playLine, args2);
  scheduler.make_runnable(q);

  scheduler.exit();
}
```

6 Current and Future Additional Work

We are currently adding facilities that will make rhythmic specification in a `SG_PROCESS` more flexible. `DUR` (duration) objects define a `NOTE` duration. `DUR(4)` is a quarter note as is `DUR(1,4)`. `REST` objects are like `DUR` objects, except they define a rest. `DUR` and `REST` objects may be inserted into `CADENCE` objects. `CADENCE` objects are composed of `DUR` or other `CADENCE` objects and they may be used to operate on sets of rhythmic figures. Therefore, one may write:

```
    // add quarter and eight note to cadence
    CADENCE < DUR(4) < DUR(1,8)
```

C++ operators are defined on `CADENCE` objects so that rhythm can be manipulated in interesting ways. For example, `CADENCE::operator<<(DUR&)` and `CADENCE::operator>>(DUR&)` shifts the rhythmic figures defined by the `CADENCE` either forward or backwards in time. This is useful for musical sections that are either *behind* or *in front of* the beat. If `c1` and `c2` are `CADENCE` objects, `c1 && c2` is the intersection, `c1`

`||` `c2` is the union, and `c1 ^ c2` is the mutual exclusion of the rhythmic figures defined by both cadences. `CADENCE::operator!()` transforms all `REST` (`DUR`) objects into `DUR` (`REST`) objects of the same time length. `CADENCE::operator()` is used to define `CADENCE` objects relative to other `CADENCE` objects. This is useful for defining rhythmic structures commonly seen in a Frank Zappa score. Common constant `DUR` objects are also pre-defined. For example:

```
CADENCE c1,c2;
// load c1 with two quarter notes and a quarter note triplet.
c1 < DUR(4) < DUR(4) < DUR(6) < DUR(6) < DUR(6);
// load c2 with two half notes.
c2 < DUR(2) < DUR(4) < DUR(4);

c1 << DUR(1,128)   // shift rhythm early in cadence
c1 >> DIR(1,128)   // shift late.

SG < c1 && c2;  // intersection
SG < c1 || c2;  // union

CADENCE c3,c4;
c3 < DUR(6) < DUR(6) < DUR(6) < DUR(6) < DUR(6) < DUR(6);
c4 < DUR(5) < DUR(5) < DUR(5) < DUR(5) < DUR(5) ; // 5 per measure
SG < c1(2,4,c2);  // return CADENCE defining 5 evenly spaced rhythmic
                      // figures given by c2 in the time frame between
                      // the 2nd and 4th component of c1
```

We have plans for further MOOD development. Good synthesizer management and music output objects need to be designed so that MOOD will understand and simultaneously use different types of synthesizers and DSP systems. This will involve building subclasses of `TEMPERAMENT` for specific synthesizers, and subclasses of `NOTE_PLAYER` for different sound modules. This will enable, for example, the NeXT DSP or the SPARCstation audio output device to be used together with MIDI synthesizers.

Extensions need to be made for more sophisticated algorithmic music generation. For example, automatic harmonizer objects are planned as subclasses of `OP_GENERATOR`. Other subclasses of `GENERATOR` could be used to build musical data structures such as the TTREE [Die88].

7 Related Work

Several computer languages for music are related to MOOD. FORMULA [AK90], based on Forth, is a real-time language for algorithmic music composition. MOOD's multiple tasks and scheduling policies are derived from FORMULA. MOOD provides a cleaner and more intuitive syntax for algorithmic music specification. Also, since MOOD is written in an object oriented language, it is believed that MOOD is more easily extendible. MOOD is also integrable into other C++ user interface packages that desire musical features. MOOD, however, currently lacks an interactive environment which FORMULA uses extensively.

The Canon score language [Dan89] for computer music, written in LISP, emphasizes nesting scores and operations (or transformations) on scores. Canon provides many interesting transformations on scores that allow musical constructs to be defined and later manipulated. Like MOOD, Canon Scores are not note lists, but are themselves programs. Both MOOD and Canon can express "complex parameterized behaviors" even though Canon is written in a declarative language and MOOD in an imperative language. Users of MOOD may define abstract parameterized musical constructs either using inheritance or by writing a procedure. Canon does not have as concise a syntax as MOOD, nor does it support multiple processes or abstraction through inheritance.

The NeXT Music Kit [JB89] is an Objective-C [NeX] musical interface to the NeXT machine. It provides a large assortment of well thought out tools one needs to build sequencers and sound editors, has an interface to play notes on the NeXT DSP chip, and is object-oriented and therefore extensible. The Music Kit's syntax, however, is unwieldy (primarily due to Objective-C) and common operations on notes are not predefined as they are in MOOD. Also, the Music Kit does not provide multiple processes.

8 Conclusion

MOOD provides a powerful language base for algorithmic computer music. Unlike musical "little languages" [Lan90], MOOD provides a uniform, extensible, and familiar environment for algorithmic music composition.

The features of C++ have contributed in many ways to the MOOD design. Inheritance allows us to define a graded set of process types, and to separate scheduling policies from the entities being scheduled. It also provides a framework in different implementations of a given interface (e.g., `NOTE_MAKER`, `NOTE_PLAYER`, and `TIME` can be easily substituted). The ability to overload many operators enables us to provide a rich and concise syntax for expressing pitch structures. The availability of C++ toolkits such as Interviews [LCV87] will facilitate the integration of MOOD with a graphical user interface.

An interactive language system (like Lisp, Forth, or Self [US87]) is often useful for computer music, and most current C++ implementations are non-interactive. A second inconvenience of C++ for our purposes is that one cannot add new control structures. This precludes language features such as FORMULA's "embedded process definitions". The advantages of C++, however, outweigh these disadvantages.

MOOD currently runs on the MC68020 and SPARC lines of Sun workstations under SunOS version 3.5 through 4.1 and has been compiled both with `g++` (the GNU C++ compiler) and AT&T Cfront 2.0. MOOD has also been ported to the Macintosh under MPW C++. We plan to port MOOD to the NeXT Machine, MIPS computers, and the IBM PC and PS/2.

9 Acknowledgements

Ron Kuivila and George Homsy both contributed to the design and implementation of MOOD. Steven McCanne wrote the UNIX real-time MIDI device driver for the SPARC-Station.

References

[AK90] David P. Anderson and Ron J. Kuivila. A System for Computer Music Performance. *Transaction on Computer Systems*, 8(1):56–82, 1990.

[Dan89] Roger B. Dannenberg. The Canon Score Language. *Computer Music Journal*, 13(1), Spring 1989.

[Die88] Glendon Diener. TTrees: An Active Data Structure for Computer Music. In *Proc. International Computer Music Conference*, pages 184–188. Computer Music Association, 1988.

[ES90] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, 1990.

[Int89] The International MIDI Association, 5316 W. 57th St., Los Angeles, CA 90056. *MIDI 1.0 Detailed Specification, Document Version 4.1*, 1989.

[JB89] David Jaffe and Lee Boynton. An Overview of the Sound and Music Kits for the NeXT. *Computer Music Journal*, 13(2), Summer 1989.

[Lan90] Peter S. Langston. Little Languages for Music. *Computing Systems*, 1990.

[LCV87] Mark A. Linton, Paul R. Calder, and John M. Vlissides. Interviews: A C++ Graphical Interface Toolkit. In *USENIX C++ Workshop Proceedings*, 1987.

[NeX] NeXT, Inc., 3475 Deer Creek Road, Palo Alto, CA 94394. *Object-Oriented Programming and Objective-C*. NeXT Technical Documentation: Appendices.

[Pop89] Stephen Travis Pope. Machine Tongues XI: Object-Oriented Software Design. *Computer Music Journal*, 13(2), Summer 1989.

[US87] David Ungar and Randall B. Smith. Self: The Power of Simplicity. In *OOPSLA'87 Conference Proceedings*, 1987.