Recent Developments in the CMU Midi Toolkit[1]

Roger B. Dannenberg

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
USA
email: dannenberg@cs.cmu.edu

1. Introduction to the CMU Midi Toolkit

The CMU Midi Toolkit is a software system that provides an easy-to-use interface to Midi, a standard interface for music synthesizers (Loy, 1985). The intent of the system is to support experimental computer music composition, research, and education (Dannenberg, 1984), rather than to compete with commercial music systems. The CMU Midi Toolkit is especially useful for building interactive real-time systems (Chabot, 1986) and for creating non-traditional scores, which might include non-standard tunings, mulitiple tempi, or time-based rather than beat-based notation. (Dannenberg, 1986)

1.1. Adagio: A Text-Based Score Language

One of the main components of the CMU Midi Toolkit is the score language *Adagio*. Unlike the numerous screen-based sequencers now available on personal computers, Adagio is a text-based system. Adagio tends to be more flexible and open-ended than most sequencers. One attractive feature is that Adagio scores can be generated by other programs quite easily. In addition, any text editor can be used to create and modify Adagio scores. Text editors with macro facilities can often be programmed to perform fairly complex transformations on scores. Adagio can also be incorporated into specialized application programs and interactive compositions.

To give some sense of what Adagio scores are like, here is a very short example:

```
* Adagio example
T100 Cs4 q Lmp Z15
D4 U40
```

The first line, beginning with an asterisk, is a comment, which is ignored by Adagio. The second line says to play a C-sharp above middle C (`Cs4`) with a duration of a quarter note (`q`), loudness *mezzo-piano* (`Lmp`), with Midi program 15 (`Z15`), and starting at time 1.00 second (`T100`). The

[1]Published as: Dannenberg, ‘‘Recent Developments in the CMU Midi Toolkit,’’ ***Ano 2000 Symposium Proceedings***, Julio Estrada, ed., University of Mexico, 1991, pp 52-62.

last line says to play a D (`D4`) above middle C with a duration of 0.40 seconds (`U40`). By default, this note will start when the previous note finishes and will have the same loudness as the previous note. There are many more details, but this short example should give the reader a good idea of how music is described in Adagio.

1.2. Moxc: Real-Time Programming Support

Another important part of the CMU Midi Toolkit is the *Moxc* programming environment, which supports the development of real-time interactive Midi-based software. Programs written using Moxc are *event-driven*: each incoming Midi event causes some action to be performed. The execution of a routine can be deferred to the future using the special function `cause`. The following example illustrates the core of a program that delays and transposes incoming Midi:

```
keydown(c, k, v) {
  cause(100,midi_note,c,k+10,v);
}
```

The `keydown` routine is called by the system whenever a Midi note-on message is received. The parameters are the channel, pitch, and velocity of the note-on message. The body of the `keydown` routine schedules a future action. The action will take place 1.00 seconds in the future and consist of a call to the routine `midi_note`. The parameters passed to `midi_note` will be the channel `c`, the transposed pitch `k+10`, and the velocity `v`. Another routine, `keyup`, is required to handle Midi note-off messages.

There is an important technique that can be used to simulate tasks within Moxc. Consider the following example:

```
play_forever(pitch)
{
    midi_note(1,pitch,100);
    cause(100,play_forever);
}
```

The `play_forever` routine turns on a note and then uses `cause` to call itself in the future. When `play_forever` is called again in the future, it will turn on the note and schedule itself once again. (This example ignores the potential problem of turning on notes without turning them off.) The effect is that `play_forever` is a periodic task that runs every 1.00 second. Multiple calls to `play_forever` could be used to interleave different pitches. This basic technique, in which a routine reschedules itself in the future, can be elaborated by passing more parameters, by computing the delay to the next execution, and by adding tests that stop the rescheduling. Overall, Moxc greatly simplifies the writing of interactive music software. Moxc eliminates many of the difficult aspects of real-time programming by managing input events, scheduling future events, and providing a simple form of task.

1.3. Transcribe and Record Functions

Two programs supplement the Adagio sequencer. The *Transcribe* program takes real-time Midi input and outputs an Adagio score file. *Record* combines the recording function of Transcribe with the playback function of Adagio, allowing the user to ‘‘overdub’’ new tracks of Midi data. There is no explicit management of tracks or merging of score files; however, scores can be merged simply by concatenating them with a standard operating system command such as ‘‘cat’’ in Unix or ‘‘copy’’ in DOS.

1.4. Exget and Exput

The CMU Midi Toolkit has several other utilities, including the Exget and Exput programs, which allow the saving and restoring of Midi System Exclusive messages. These messages are mostly used for loading and storing patch information. Exget and Exput are written to be synthesizer independent.

The CMU Midi Toolkit has been widely used for teaching, research, and composition, and it has been distributed to hundreds of users, but experience has called attention to a number of shortcomings. In the following sections, I will discuss additions that have been made to the basic framework and explain briefly why these extensions are useful.

2. New Moxc Features

The major change in the CMU Midi Toolkit is the concept of multiple virtual time systems. A virtual time system provides a time reference against which events are scheduled. Unlike real time, which by definition progresses at a fixed rate, virtual time is under program control. Thus, if a sequence of events is scheduled relative to a virtual time system, the events can be made faster simply by setting the virtual time system to run faster.

In Moxc, virtual time is represented by what is called a *timebase*. Any number of independent timebases can be created as shown in this example:

```
timebase_type slowbase =
        timebase_create();
timebase_use(slowbase);
cause(100, some_activity);
```

Here, a timebase named `slowbase` is created. The second statement tells Moxc to schedule events relative to `slowbase`, and the last line starts `some_activity` at (virtual time) 100. The timebase can be manipulated by various operations, including setting the ratio of real time to virtual time and setting the virtual time. Virtual time can also be stopped:

```
set_rate(slowbase, 2 << 8); /* half speed: */
set_virttime(slowbase, 2000);
set_rate(slowbase, STOPRATE);
```

Each timebase maintains a queue of events that are scheduled for the future. (Dannenberg, 1989a) The queue is maintained using the heapsort algorithm (Aho, 1974) which is much faster than the linear search employed in the previous version of Moxc.

Time is now maintained with a resolution of 1ms versus the 10ms time unit of the previous version of Moxc. To maintain compatibility, the default timebase runs at 1/10 realtime, making the default unit of virtual time be 10ms.

2.1. Sequences

The main reason for introducing the timebase was to allow multiple sequences to be started, stopped, and controlled conveniently. Moxc can now load and play multiple Adagio scores. The following example creates a sequence object, loads sequence data from a file, and plays the sequence:

```
    seq_type my_seq = seq_create();
    seq_read(my_seq, fp);
    seq_play(my_seq);
```

It is possible to create any number of sequences and play them independently within a single Moxc program.

Sequences have a number of parameters that can be set under software control. The following statements show how to transpose a score, change the rate of playback, offset the Midi note-on velocities, enable or disable individual channels, and cause a routine to be called when the sequence finishes:

```
    seq_set_transpose(my_seq, 3);
    seq_set_rate(my_seq, 200);
    seq_set_level(my_seq, -10);
    seq_set_channel_mask(my_seq, 3);
    seq_at_end(my_seq, stop_action);
```

In addition to these functions, it is possible to start playing a sequence at any time point and control the virtual time against which events are scheduled.

3. New Adagio Features

This section describes new features of Adagio, some of which enable an even closer coupling between Adagio and Moxc.

3.1. Midi Synchronization

Adagio predates Midi, so there are a number of concepts in Midi that were not supported by the original Midi version of Adagio. One of these is the real-time clock feature that enables Midi devices to synchronize to a ‘‘master’’ device. Since Adagio supports multiple tempi, and Midi clock is based on beats, it is necessary to be explicit in the score about where the clock should start and what is the duration of a quarter note. The `!clock` command in Adagio turns on a 24 pulse-per-quarter (PPQ) clock at the current tempo and time:

```
    !clock
```

A `!clock` command must be inserted for each tempo change that is to be reflected in the Midi clock.

The Adagio program can act as either a master or slave. As a master, Midi real time messages are generated according to the score. As a slave, the program uses incoming Midi messages to control the timebase against which notes and other events are scheduled. In this way, an external device such as a drum machine or other sequencer can start, stop, and control the tempo of Adagio scores.

3.2. System Exclusive Messages

Adagio now has a definition facility that makes it convenient to send system exclusive parameters. Often, there are parameters on Midi synthesizers that can only be controlled by system exclusive messages. Examples include the FM ratio and LFO rate on a DX7 synthesizer. The following example defines a macro for the DX7 LFO rate and then shows how the macro is used to set the LFO rate for a B-flat whole note in the score. The macro definition is given in hexadecimal, except ``v'' is replaced by the channel (voice) and ``%1'' is replaced by the first parameter:

```
!def lfo F0 43 0v 01 09 %1 F7
Bf5 W ~lfo(25)
```

3.3. Control Ramps

One of the problems with the first Adagio implementation was that each control change message had to be specified individually. The new implementation provides a `!ramp` command that specified a smooth control change from one value to another. In addition, any control change can be specified using the syntax ``~N(V)'', where N is the controller number, and V is the value:

```
!ramp ~23(10) ~23(50) U20 W
!ramp ~lfo(15) ~lfo(35) U10
```

The first line says to ramp controller number 23 from value 10 to value 50, sending a new control change message every 20 time units. The overall duration of the ramp should be equivalent to a whole note (`W`). As shown in the second line, even parameters controlled by system exclusive messages can be specified.

3.4. Time Units and Resolution

The default time unit remains at 10ms (ten milliseconds or one centisecond), but since the basic time unit of Moxc was changed to milliseconds, it is possible to change the units in Adagio as well. The time unit can be specified by the `!csec` or `!msec` commands.

In order to work with smaller time units and larger spans of time, the computation of time now uses 48-bit arithmetic, and note durations are accumulated with 8-bit fractions, giving an internal accuracy of about 8 microseconds. This is necessary because the cumulative effect of roundoff errors could otherwise become significant.

3.5. Midi File Reader

The new version of Adagio can read and play standard Midi files. This means that Midi files can also be loaded and played by Moxc programs.

3.6. Calling C Routines

Once it became possible to load Adagio sequences into Moxc programs, it became natural to look for a way to call Moxc routines from within Adagio scores. For example, a trill can be expressed computationally in Moxc, but the natural place to indicate a trill would be in an Adagio score. This is supported by the `!call` command in Adagio, which calls a C routine that can in turn invoke a complex sequence of operations. Below is a call to a trill routine. Notice that Adagio notation can be used to indicate the pitch of the trill `A5`, the whole-note duration `W`, and the loudness `mf`:

```
!call trill(A5,W,2,S,mf)
```

Another use of the `!call` command is to simulate typed commands from the score. This allows interactive programs to be used for testing or rehearsal purposes. Later, the interactive commands (such as typing ‘‘x’’) can be automated by placing them in a score:

```
    !call asciievent('x') T400
```

Finally, scores can be used to call upon and structure very high-level compositional algorithms written in Moxc:

```
    !call melody(C3,C5,W4,Lmp)
```

3.7. Setting C Variables

In addition to calling Moxc routines, there is another way in which scores can control Moxc programs. The `!seti` command sets an integer variable to a value, and the `!setv` command sets an element of an integer array:

```
    !seti delay 200
    !setv transposition 5 -4
```

As with the `!call` command, these commands perform their operations at particular times according to their place in the Adagio score. This makes it very easy to implement time-varying parameters that control various aspects of an interactive music system.

4. An Example

So far, the largest application using the new version of the CMU Midi Toolkit is my composition *Ritual of the Science Makers*, for three live musicians, computer music system and computer animation. My goal in composing *Ritual* was to use a computer to amplify and transform the live performance of a small ensemble. I want a strong connection between the actions of the performers and the sounds and images generated by the computer. This is achieved by interfacing the instruments, via Midi, to the computer system. Rather than write a score for the computer, a Moxc program responds to the incoming Midi messages by generating sequences of musical and animation material. (Dannenberg, 1989b)

One of the problems with such interactive compositions is that it can be very difficult to sustain interest in a work because the computer’s response is always generated by the same algorithm. For *Ritual*, I defined about ten sections, each of which reacts differently to the performers, and each of which is associated with different graphical images. Because there are so many sections, there are a large number of parameters that need to be changed to make a transition from one section to another: synthesizer patches are selected, levels are adjusted, reverb and delay times are altered, and internal program variables are set. All of this is performed by Adagio scores (one per section) which are started by typing the section number at the computer console.

In one case, I used Adagio to send Midi notes directly to synthesizers because I wanted a particular passage played with the live performers. In most cases, however, Adagio is only used to set parameters on command. Even this turned out to be exceedingly valuable; in rehearsals, I can jump to any section of the piece just by typing the section number. Adagio makes it relatively easy to change performance parameters because the scores can be changed without recompiling or linking the program. Another advantage is that many parameter changes must take place smoothly over time. Adagio is ideal for notating these transitions.

In a few cases, it was not practical to control parameters directly from Adagio. For example, there is one section where I want to change program variables over time according to a fairly complex trajectory. To accomplish this, I use the `!call` command to invoke a Moxc process to do the work. The trajectory can still be controlled from within Adagio scores by passing parameters. Another use of the `!call` command is to chain sections together: instead of manually starting section 3, I can write `!call ascii_event('3')` in the score at the end of section 2.

5. Conclusions

The CMU Midi Toolkit supports the development of interactive music systems. The recent changes have allowed a blending of the computational model supported by Moxc and the sequence model supported by Adagio. Now, sequences can be initiated under program control, and computations can be invoked from within a sequence. The interactive computer music composition *Ritual of the Science Makers* uses both computation and stored sequences to respond to live instruments. The CMU Midi Toolkit is currently being used to develop conducting and score-following software (Bloch, 1985, Dannenberg, 1988).

6. Acknowledgments

The CMU Midi Toolkit has been developed with the help of Dean Rubine, John Maloney, Jean-Christophe Dhellemmes, Joe Newcomer, and George Logeman. The CMU College of Fine Arts and School of Computer Science deserve credit for their support of computer music, without which this work would not have been possible. *Ritual of the Science Makers* was written in honor of the 25th Anniversary of Computer Science at CMU.

References

Aho, Hopcroft, and Ullman. *The Design and Analysis of Computer Algorithms*. Addison Wesley, 1974.

Bloch, J. J. and R. B. Dannenberg. Real-Time Computer Accompaniment of Keyboard Performances. In B. Truax (Ed.), *Proceedings of the International Computer Music Conference 1985*. International Computer Music Association, 1985.

Chabot, Xavier, Roger Dannenberg, and Georges Bloch. A Workstation in Live Performance: Composed Improvisation. In P. Berg (Ed.), *Proceedings of the International Computer Music Conference 1986*. International Computer Music Association, 1986.

Dannenberg, F. K., R. B. Dannenberg, and P. Miller. Teaching Programming to Musicians. In D. Mansfield (Ed.), *Proceedings Fourth Symposium on Small Computers in the Arts*. Washington, D.C.: IEEE Computer Society, 1984.

Dannenberg, R. B. The CMU MIDI Toolkit. In *Proceedings of the 1986 International Computer Music Conference*. San Francisco: International Computer Music Association, 1986.

Dannenberg, R. B. and H. Mukaino. New Techniques for Enhanced Quality of Computer Accompaniment. In C. Lischka and J. Fritsch (Ed.), *Proceedings of the 14th International Computer Music Conference*. San Francisco: International Computer Music Association, 1988.

Dannenberg, Roger B. Real-Time Scheduling and Computer Accompaniment. In Mathews, M. V. and J. R. Pierce (Ed.), *System Development Foundation Benchmark Series. Current Directions in Computer Music Research*. MIT Press, 1989.

Dannenberg, R. B. Real Time Control For Interactive Computer Music and Animation. In N. Zahler (Ed.), *The Arts and Technology II: A Symposium*. New London, Conn.: Connecticut College, 1989.

Loy, G. Musicians Make a Standard: The MIDI Phenomenon. *Computer Music Journal*, Winter 1985, *9*(4), 8-26.