PMML: A Music Description Language Supporting Algorithmic Representation of Musical Expression

Satoshi Nishimura
School of Computer Science and Engineering
The University of Aizu
Aizu-Wakamatsu city, Fukushima, 965-8580, Japan
Email: nisi@u-aizu.ac.jp

Abstract
This paper describes a novel music description language called the Practical Music Macro Language (PMML), intended for the computer-controlled performance of expressive music using MIDI instruments. A remarkable feature of the PMML is its ability to specify expressive parameters algorithmically. A toolkit consisting of a compiler which translates a PMML source code to the Standard MIDI file, a discompiler, and an Emacs-based data entry tool is developed.

1 Introduction
So far many music description languages have been proposed; however, little attention has been paid to the representation of musical expression which is usually not designated on scores. Although most languages have means for specifying expressive parameters (e.g., dynamics and tempo fluctuation) on the basis of a single note or a group of notes, it is too tiresome to describe expressive music only with these functions. Some languages such as Formula [AK91] enable us to specify those parameters according to some graphical shapes; however, we still have many things which must be done manually. For example, in order to increase the loudness of the highest-pitch note for each chord, we need to designate the velocity increment chord by chord.

The PMML is a novel MIDI-based event description language addressing the aforementioned problem by supporting the procedure-based representation of musical expression. The syntax of the PMML is similar to that of the C/C++ language which would be today's most common programming language. In the PMML, most features provided in existing music languages are supported; for example, concurrent processes as found in Pla [Sch83], FORMES [RC84], and Formula, the representation of musical structures and part hierarchy, algorithmic composition, procedure-based note parameter conversion as in SCORE [Smi72], and the specification of note parameters according to connected line segments or free-form curves. In addition, the PMML provides methods for defining and applying procedural rules for making music expressive. For instance, users can define a rule of emphasizing the top note of each chord as explained above, adding extra velocity for syncopated notes, or applying crescendo/decrescendo for ascending/descending note sequences. Those rules can be applied to any temporal span of music for any set of voice parts with simple directions embedded in music description.

2 Basic Features of the PMML
The method for representing musical phrases in the PMML is similar to MML [Mat97], which is diffused mainly in Japan for controlling the sound generators of personal computers. In the PMML, a note or rest is represented by a single alphabetical letter prefixed by an arbitrary number of octave adjusters ('s or '_') and followed by an arbitrary number of accidentals (#'s or b's). For example, an ascending D-major scale is described by:

```
D E F# G A B _C# _D
```

Note parameters other than pitch are derived from a context, in which duration, velocity (or loudness) and other note attribute values are maintained. The context can be varied at any time with various context-modifying directives. For example, an 'h' letter changes the duration value in the current context to a half
note, effectively specifying that its succeeding notes are half notes. In the PMML, chords are described by putting some notes in brackets (e.g., '[C E G]').

The PMML is a multi-threaded language, where threads are hierarchically organized in a tree structure. Each thread has its own copy of the context. At the time of thread generation, the context of a thread is duplicated from its parent thread. Threads are categorized into two types: synchronous threads and asynchronous threads. After a thread is generated, its parent waits for the thread to terminate if the thread is synchronous; otherwise, the parent does not wait. Synchronous threads are used for altering contexts temporarily, while asynchronous threads are utilized for describing multiple voice parts.

3 Direct Specification of Musical Expression

The PMML enables us to specify expressive parameters non-algorithmically with a minimum amount of description. This feature can be used in conjunction with the algorithmic representation of musical expression. The following code is an example for changing velocity for a group of notes:

```
C D E (v=100) F G A B -C
```

In this example, the velocity of the first three notes is 80, the default value of the PMML. The braces create a child synchronous thread and evaluate the descriptions in between in the child thread. The 'v=100' directive changes the velocity value in the child thread’s context to 100; as a result, the velocity of the F and G notes becomes 100. The velocity of the last three notes is again 80, because they are evaluated in the parent thread. We can temporarily change other note attributes (e.g., duration and delay time) in the same way.

To specify expressive parameters on a per-note basis, postfix notations as below are also available:

```
C D E F(v=120) G(v+=20) A B ^C
```

In this example, the velocity of the F note is 120, and that of the G note is 100 (i.e., 20 units higher than the default value). Synchronous threads are created for each of those notes.

4 Algorithmic Representation of Musical Expression

To specify musical expression algorithmically, the PMML utilizes an event-processing module called an effector. The effector invokes a user-definable procedure called an action for each event in a particular segment of music. Effectors are especially useful for the algorithmic representation of musical expression, but also applicable to various music manipulation such as pitch conversion, time deformation, and chord arpeggiation.

The following code is an example of the definition of an effector, which increases the velocity of each note by the double of the pitch difference from the middle C and thus applies crescendo/decrescendo to ascending/descending note sequences in a simple way:

```python
def eff(autoswell) { // Define an effector named 'autoswell'
    case(note) { // Action for note events
        v += (n - C4) * 2 // Modify velocity according to the note number
    }
}
```

The above code merely defines the behavior of the effector, and note velocity is not affected until the effector is attached to a thread. Putting the effector name with optional arguments attaches the effector to the current thread. The context of each thread contains a list of attached effectors, and like other components in the context, the list is copied from the parent at the time of thread creation. Consequently, the valid span of effector attachment can be indicated using braces or parentheses in the same way as the temporary change of velocity explained in Section 3. For example, the following code applies the 'autoswell' effector only to the notes between the braces:

```
C D E {autoswell} C D E F G F E} D C
```
def eff(slim_chord, "n") { // "n" means a numeric argument.
    init { // action executed once before event processing
        $prev_note = #() // initialise the one note buffer
    }

    case(note) { // action executed for each note
        property = #(<t, v, a, h>, )
        if( !$prev_note ) { // reject // do not output the note at the beginning
            $save = #(<t, v, a, h>)
        } else { // replace the current note with the previous one
            $v = $prev_note[3] // set the velocity
            if( t == $prev_note[1] ) { // If the onset time is the
                $v -= 1 // same as the previous note,
                $t -= 1 // decrement the velocity.
            }
        }
    }

    wrap { // action executed once after event processing
        // flush the buffer contents
    }
}

Figure 1: Effector for Trimming Chord Loudness

Figure 2: Effector Library

Figure 1 shows the definition of the 'slim_chord' effector which decrements the velocity of notes except the last-designated note (usually having the highest pitch) in each chord. This is more frequently used than an effector incrementing the velocity of the last-designated note. The amount of the decrement is passed to the effector as an argument when the effector is attached. Since effector actions are invoked in the chronological order of note onset time, the above function is fulfilled by decreasing velocity for such a note that its next note has the same onset time. In the effector shown in Figure 1, a buffer for storing a note is used, and the onset time is compared with that of the previous note instead of the next note.

Figure 2 lists the currently available system-defined effectors. With these effectors, users can describe expressive music with minimal effort.

5 Example of the Description of Musical Expression

In practice, music expression is described by a combination of direct specification, algorithmic specification, and shape-based control along connected line segments or free-form curves. Figure 3 represents measures 53 and 54 of Grieg's piano concerto with expressive control of velocity, whose fluctuation is displayed in Figure 4 with a dot for each note. In this example, the note velocity is formulated as \( (80 - d + r - c) \times m \), where \( d \) is the decrement represented by the '--' sign, \( r \) is a random value between -3 and 3, \( c \) is the decrement imposed by the 'slim_chord' effector, and \( m \) is the value on a piecewise-polynomial curve specified by the three points designated in the source text. The dotted line in Figure 4 indicates the value of \( 80 \times m \).

Figure 3: A Tune from Grieg's Piano Concerto

Figure 4: Velocity Fluctuation of Figure 3
6 Tools

I developed the following language-processing tools for the PMML. Some of these tools are publicly available at our FTP site (ftp://ftp.u-aizu.ac.jp/u-aizu/pmml/).

**Compiler** — This translates a PMML source code to a Standard MIDI file. The compiler consists of over 18,000 lines of C program codes. The time needed for the translation is typically less than one second, and thus, composers can repeat the cycles of editing, retranslating, and playing comfortably.

**Discompiler** — This converts a Standard MIDI file to a PMML source code. The discompiler outputs either an event or measure for each line depending on the user’s preference. For inexpressive (or mechanical) data, the discompiler produces sufficiently readable output.

**Language-specific editor** — This is a tool for data entry and music editing based on the GNU Emacs editor (Figure 5). It consists of an Emacs Lisp code and a server program interfacing with MIDI synthesizers. Pitch names can be typed in with a musical keyboard, which enables us significantly quicker data entry than using an alphanumeric keyboard. Playing described music by invoking the compiler and then sending the compiled result to the server program is accomplished by a few key strokes. It is also possible to play music from somewhere in the middle of a phrase indicated by the current cursor position. Recording real-time performance and then inserting its discompiled result to the source text are also attainable with simple operations. In the future, we would like to add a piano-roll display to this tool.

7 Related work

The notion of an event processing module is also found in other languages; the earliest one is SCORE [Smi72] in which FORTRAN subroutines can be used to perform some actions for each note. The phrase processor of Flavors Band [Fry84] is much closer to the effector of the PMML. However, these languages lack a mechanism for attaching such a module to a thread; and therefore, the temporal/spatial span to which the module is applied can not be specified so easily as the PMML.

8 Summary

This paper proposed a novel music description language which is able to specify expressive parameters algorithmically. By utilizing this feature as well as various libraries, even a user who is not proficient in performing instruments can generate expressive music with little effort.

References


