音楽情報科学 21−10
(1997. 7. 21)

音楽記述言語 PMML の概要

西村 憲
会津大学 コンピュータ理工学部

本稿では、PMML (Practical Music Macro Language) と呼ばれる MIDI 楽器の自動演奏を目的とした新たな音楽記述言語について概説する。PMML は、音符、休符、和音、並行する複数のパート等の基本的な音楽要素を簡便に記述する機能に加え、自由曲線に基づいたパラメータの連続変化、制御構造やマクロを使った音楽の構造記述、アルゴリズムによる楽曲合成、エフェクタと呼ばれるソフトウェアモジュールによるイベント処理、メッセージパッシングによるパート間での通信または同期といった、より高度な機能を備えている。PMML のソースコードは開発した PMML インタプリタによって標準 MIDI ファイルに変換される。この変換に要する時間は、ソースコードの編集、再変換、演奏を繰り返し実行する上で十分実用的な値である。

Overview of the Practical Music Macro Language

Satoshi Nishimura
The University of Aizu

Abstract

This paper gives an overview of a novel music description language called the Practical Music Macro Language (PMML), which is intended for the computer-controlled performance of MIDI instruments. The PMML enables us concise description of basic musical elements such as notes, rests, chords, and concurrent voice parts. In addition, the PMML supports more advanced music-describing methods such as continuous parameter change based on free-form curves, structural representation of music with control structures and macros, algorithmic composition, event processing with a software module called an effector, and communication/synchronization among voice parts with message passing. A PMML interpreter which translates a PMML source code to a standard MIDI file is developed. The time for the translation is practical for repeating the cycles of editing, retranslating, and playing.

1 はじめに

音楽記述言語は、電子楽器を対象とした音楽の制作を効率的に行う上で重要な役割を果たす。特に、既存の楽曲またはその部分的要素をデータベース化し、それらの再利用を促進するという効果が大きい。また、音楽記述言語の利用によって、通信ネットワークを介した音楽情報の転送において、標準 MIDI ファイルのようにイベント列だけのデータを転送する場合に比べ、より高水準の情報（例えば繰り返し構造など）を伝えることができる。

本稿で扱う音楽記述言語は、イベント記述を対象とした演奏指向言語であり、従って、MIDI シンセサイザへ送出するイベントの列をテキスト形式で高い抽象度をもって表現することを第一の目的とする。楽譜清書や波形合成、さらに入力装置からのイベントをリアルタイムに処理するといった用途はここでは考えない。ただし、将来はこれらをサポートするように現在の言語を拡張したいと考えている。

イベント記述を対象とした演奏指向言語は、作曲や編曲を行う上で実用的でなければならない。このための条件として、以下のようなものが挙げられる。

(1) 逐次的な音符・休符の列、和音や並行する複数のパートなどの基本的な音楽要素を、少ないタイプ量で、しかも高い可読性をもった形で表現できること。

(2) アクセント、クレッシェンド、テンポルバートな

どの音楽的表現を容易に記述できること。

(3) 高水準の記述が可能であること。特に、繰り返し、パターン呼び出し等、音楽の構造を表現できるとともに、アルゴリズムによる楽曲合成が可能であること。

(4) 言語処理時間が短く、ソースコードの編集終了から演奏までの待ち時間が実用的な値であること。

過去に多くのイベント記述言語が提案されているが、上の条件をすべて満足するものはない。パソコン用BASIC言語の一部として実装されているMML [4]は(1)の点において、タイプ量に関しては優れているが、可読性は低く、また(2)の点においても不利である。CMU MIDI Toolkit [3]のAdagioもタイプ量は少ないが、(2), (3)の点が考慮されていない。AndersonとKuivilaによって提案されているFormula [2]は、(1)の点において、並行プロセスの概念を導入することによって和音や並行パートの系統的な扱いを可能としたが、可読性にはあまり重点をおいていない。また、このFormulaや、Pla [5]、Common Music [6]などのように汎用プログラム言語を拡張したものは、(3)の点で万能であるが、(1)や(4)の点で音楽専用言語に比べると必然的に劣ってしまう。

筆者は、このような背景をもとに、上に挙げた条件をすべて考慮した言語であるPMML (Practical Music Macro Language)を提案する。本稿においては、PMMLの概略のみを説明する。言語仕様の詳細については[7]を参照していただきたい。

2 基本的な音楽要素の記述

本章では、音符・休符の列、和音、並行する声部などの基本的な音楽要素の記述について述べ、更にベロシティ、音の持続時間など音符パラメタの指定方法について説明する。

2.1 スレッド

PMMLでは1つ以上の動的に生成・消滅するスレッドを用いて音楽を記述する。スレッドには名前つきスレッドと名前なしスレッドの2種類がある。各スレッドは表1に示すレジスタをそれぞれ独立に持っている。

スレッドには親子関係があり、すべてのスレッドは1つの木構造を形成する。新たに生成されたスレッドはそれを生成したスレッドの子になり、その際すべてのレジスタ値は親スレッドよりコピーされる。木の根に位置するスレッドはルートスレッドと呼ばれ、初期状態では唯一これだけが存在する。

表1: PMMLの主なレジスタ

t	イベント生成時刻（発音開始時刻）
dt	イベント生成時刻の修正値
tk	トラック番号
ch	MIDIチャネル番号
n	MIDIノート番号
tp	MIDIノート番号の修正値
o	オクターブ
key	調
v	ベロシティ
nv	ノートオフベロシティ
l	音長
do	持続時間のオフセット値
dp	持続時間の音長に対する百分率

```
// (a) A D-major scale and a rest
D E F# G A B ^C# ^D R

// (b) Specifying length and velocity
v=90      // Set velocity to 90
q         // Set length to a quarter note
D E F
w         // Set length to a whole note
G

// (c) Local change of note length
w C D {q E F} G

// (d) Use of modifiers
q C D(h) E F(v=100) G

// (e) Chords
w [D F ^C] [D F B] [E G ^C]
```

図1: 基本的な記述

表2: 音長の表記

w	全音符
h	2分音符
q	4分音符
i	8分音符
s	16分音符
z	32分音符
q.	付点4分音符
2w	二全音符
q/3	3連符(4分音符の1/3)

```
defthread(righthand, lefthand)  // create threads

righthand {    // Select "righthand" thread
  o=5 q C C G G A A G G   // Phrase1
}
lefthand {     // Select "lefthand" thread
  o=4 q _C C E C F C E C  // Phrase2
}

righthand {
  F F E E D D h C     // This phrase follows Phrase1
}
lefthand {
  D _B C _ A F G h C  // This phrase follows Phrase2
}
```

図 2: 名前つきスレッドによる複数パートの表現

2.2 音符と休符

音 (休) 符は、音符コマンドと呼ばれる、直後に任意個の臨時記号 (#またはb) を伴った単一アルファベット文字によって表現される (図 1(a))。音符のオクターブは、アルファベット文字の前に相対オクターブ指定 (^または_の列) を置くか、或は後ろにオクターブを表す数字を置くことによって指定できる。この指定が無い場合は`o` レジスタの値がそのまま使われる。

音符コマンドの実行において、発音時刻、音長 (2つの連続する音符間の発音時間の差)、音の持続時間、ベロシティ (音の強さ) などのパラメタは、現在選択されているスレッドの持つレジスタの値が使われる。レジスタの値は自由に変更することができ、それによって同一スレッドのそれ以降に現れる音 (休) 符のパラメタを指定することができる。レジスタ値の変更は "レジスタ名=式" という代入形式によって行えるが、音長を指定する`l` レジスタの場合は、表 2 に示す表記を使っても変更できる (図 1(b))。発音時刻を表す`t` レジスタは、音符コマンドの実行ごとに、`l` レジスタの値が加算され次の音 (休) 符の発音時刻を示すように修正される。

2.3 複数のパート

PMML では名前つきスレッドを利用することによって複数のパートから成る音楽を表現する。名前つきスレッドは`defthread` コマンドによって生成され、スレッドの切替えはスレッド名を前置した中括弧を用いる (図 2)。1つのスレッドに関するコマンド列は自由に分割することができ、さらにその分割したコマンド列は他のスレッドのコマンド列における任意の位置に挿入することができる。このような柔軟性により、並行して演奏されるフレーズをテキスト上の近い位置に置いてそれらの関連を見易くすることができる。

2.4 局所的なレジスタ値の変更

表情豊かな音楽を記述するには、音符ごとに 2.2節で述べたようなパラメタを調節することが必要になってくる。その際、一時的にレジスタ値を変更し、あとで元の値に戻す機能があれば、そのような調節が容易になる。PMML は子の名前なしスレッドを生成することによってこれを実現する。

名前なしスレッドの生成法には 2 通りがあり、1つはスレッド名を前置しない中括弧を用いる方法である (図 1(c))。中括弧は、新たな子の名前なしスレッドを生成し、現在のスレッドをそのスレッドに切り替え、中括弧内のコマンド列をそのスレッドにおいて実行したのち、最後にそのスレッドを消滅させて親スレッドに主導権を戻す。子スレッドでのレジスタ変更によって親スレッドのレジスタ値が影響されることは無いため、親スレッドに戻ったとき、レジスタ内容は左中括弧以前の状態に戻る (ただし`t` レジスタを除く)。従って、中括弧内でレジスタを変更すれば、それは中括弧内のそれ以降のコマンドだけに影響する局所的な指示となる。親スレッドに戻った直後の`t` レジスタの値は子スレッドが消滅する直前の`t` レジスタの値になる。これは親スレッドが子スレッドの消滅を待つことによる。

名前なしスレッドを生成するもう 1 つの方法は、修飾子と呼ばれる構文である。これは、1つの音 (休) 符だけについてレジスタ値を変更する場合に便利な構文である。音符または休符に続けて、小括弧で囲んだコマンド列を置くと、その音 (休) 符のためだけの子スレッドが生成され、小括弧内のコマンド列および音 (休) 符の生成はその子スレッドにおいて実行される (図 1(d))。従って、小括弧内にレジスタ変更コマンドがあれば、それはその音 (休) 符だけに有効となる。例えば、小括弧内に音長指定 (すなわち`l` レジスタの変更コマンド) があれば、その音 (休) 符の音長だけが一時的に変更されることになる。

ベロシティまたは発音時刻を 1 つの音符に対してだけ増減する場合には、アクセント (++, --など) および時刻シフタ (>>, <<など) という特別の修飾子が用意されている。これらは`v` と`dt` レジスタに関して小括弧を使った記述よりも簡便な局所的変更法を提供する。

2.5 和音

和音はいくつかの音符コマンドを大括弧で囲むことによって記述される (図 1(e))。大括弧は、新たな子

スレッドを生成し括弧内のコマンド列をそのスレッドにおいて実行するという点では中括弧と同じであるが、括弧内の音符コマンドを実行する際にt レジスタを前進させない点が異なる。このため、大括弧内の音符は重なって発音される。

次のように大括弧の中にさらに中括弧を置くと、和音の中で一部の声部が逐次的に動くような構造を表現できる。

```
w [D F {h ^C B}]
```

中括弧内は、通常通り音符コマンドによってt レジスタが更新され、従って逐次的な演奏が行われる。和音全体としての音長は、その構成音の音長（中括弧が内側で使われている場合はその演奏時間）のうちで最大のものと等しい。

2.6 並行する声部の表現

しばしば一つのパートが一時的に複数の声部に分かれて演奏を行うことがある。このような場合において、声部どうしが対等な関係にあるときは次のように大括弧の中に中括弧を入れた形式で表現すれば良い。

```
[{w D E} {w F G} {h ^C B w ^C}]
```

しかしこの表現法では、主声部に対して他の声部が従属的であるという関係の場合には、従属的な声部によって主声部の自然な記述が妨害されかねない。PMML では、このような用途のために'&' つきの中括弧が用意されている。例えば上の例は次のように書き換え可能である。

```
{w D E}&
{w F G}&
{h ^C B w ^C}
```

'&' つき中括弧の動作は'&' のない中括弧とほぼ同じであるが、親スレッドが子スレッドの消滅を待たない点が異なる（この意味で'&' はUnix シェルにおけるバックグランド実行指定に類似している）。従って、中括弧内のコマンド列は'}&' より後にあるコマンド列と並列に実行される。従属的な声部は、'&' つき中括弧の中に記述することによって、主声部の記述を妨げることなく容易に追加が可能である。

3 連続コントロールチェンジ

PMML にはノートオン、ノートオフ以外のMIDI メッセージ（例えば、コントロールチェンジ、プログラムチェンジ、エクスクルーシブなど）を生成するためのコマンドが多種類用意されており、音符コマンドと混在させることができる。これらのコマンドは

```
vol(0)
C D E F G A B ^C
vol_to(80)
^C B A G F E D C
vol_to(0)
```

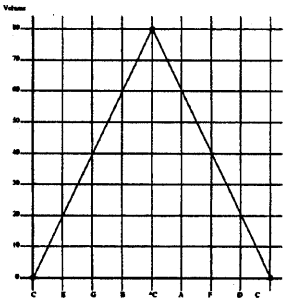

図 3: 直線に沿った連続コントロールチェンジ

```
vol(0)
C D E
vol_pt(15)
F G A
vol_pt(20)
B ^C ^C B
vol_pt(80)
A G F E
vol_pt(10)
D C
vol_cto(0)
```

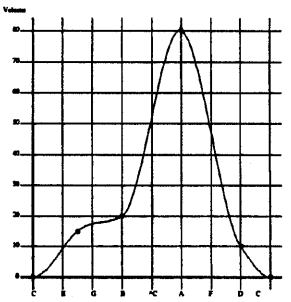

図 4: 自由曲線に沿った連続コントロールチェンジ

音符コマンドと同様に多くのデフォルトパラメタを現在選択されているスレッドのレジスタから得る。

ボリュームコントロール（7番のコントロールチェンジ）を使ったクレッシェンドなどのようにコントロール値を連続的に変化させることが必要な場合、MIDI では多数のコントロールチェンジメッセージを小刻みに出力しなくてはならない。PMML はこれを自動的に行う機能を持っている。変化の形状として、折れ線に沿ったものと、自由曲線に沿ったものの2種類が用意されている。折れ線については [2], [5] で既に報告されているが、自由曲線を用いたものはこれまで報告例がない。

図 3は折れ線による連続コントロールチェンジの使用例である。この例ではハ長調音階の上行に合わせてクレッシェンドし、下行に合わせてデクレッシェンドを行っている。このように、折れ線の最初の点の位置に対して通常のコントロールチェンジコマンド、そして以降の各頂点の位置に対してX_to（X はvol などコントローラの名称）というコマンドを挿入することによって、折れ線に沿った連続コントロールチェンジが実現できる。生成される個々のコントロールチェンジメッセージの間隔は、各コントローラ種別毎に予め定められているが、ユーザがこれを変更することも可能である。

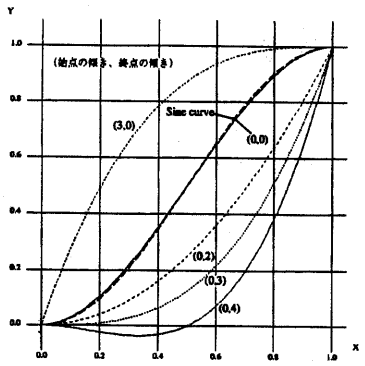

図 5: 傾き指定による曲線形状の変化

```
// Prelude C-major   J. S. Bach
def(pattern) {
  repeat(2) {
    { h $1++ }&
    { r(s) h-s $2 }&
    { r(i) s $3 $4 $5 $3 $4 $5 }
  }
}

pattern(c4, e4, g4, c5, e5)
pattern(c4, d4, a4, d5, f5)
pattern(b3, d4, g4, d5, f5)
pattern(c4, e4, g4, c5, e5)
pattern(c4, e4, a4, e5, a5)
pattern(c4, d4, f#4, a4, d5)
// to be continued
```

図 6: マクロの活用例

自由曲線の場合は、曲線の始点、終点、及び0個以上の中間点をコマンドによって指定する（図4)。これによって、始点を出発しすべての中間点を順に通って終点に達するような滑らかな曲線が想定され、それに沿ってコントロールチェンジメッセージの列が生成される。曲線の生成法として、当初、3次のスプラインを考えたが、振動や大域性という問題のため実用は困難であることがわかった。そこでここでは秋間による曲線生成法 [1] を用いた。この方法は、スプラインと同様に区間ごとに定義された3次関数に基づくが、節点での連続性は1次導関数までしか保証されていない。その代わりに振動や大域性が少ないという利点を持つ。連続コントロールチェンジにおいて2次導関数の連続性を要求することはほとんど無いため、曲線の生成法としてはこのほうが妥当である。

曲線の始点および終点における傾きは、特に指定のない場合は0となるが、コマンド引数によって指定することもできる。これを利用すれば、中間点を用いずに指数的または対数的な曲線を得ることができる（図5)。また、この図より分かるように、正弦曲線も中間点を用いずにかなりの精度で近似できる。

上で述べた連続コントロールチェンジの手法はテンポ制御にも適用できる。また、ベロシティ拡大率という仮想的なコントローラを導入することによって、ベロシティを直線または曲線に沿って変化させることも可能となっている。ベロシティ拡大率はMIDIチャネルごとに別の値を持つ内部的なコントロール値であり、実際に生成されるノートオンメッセージのベロシティは、指定されたベロシティとベロシティ拡大率の積である。

4 制御構造とマクロ

PMMLは豊富な制御構造（**if**, **while**, **repeat**, **for**, **foreach**, **switch**, **break** など）を持っている。これらによって、音楽における繰り返し構造を高い自由度をもって表現できる。少なくとも、楽譜で通常使われている繰り返し関係の指示については、楽譜でのフレーズの出現順序とテキスト上でのそれを一致させるように記述することができる。

これらの制御構造に加え、PMMLには強力なマクロ機能が備わっている。マクロは共通するフレーズや伴奏パターンを記述するために非常に有用である。マクロには変数マクロと関数マクロとが存在する。変数マクロは“マクロ名=式”という代入形式によって定義され、引数をとることはできないが高速に展開されるという特徴を持つ。関数マクロは**def** コマンドによって定義され、任意個の引数をとることができる。引数の種類としては、呼び出す時点で式の評価が行われる即時評価型とトークン列のまま渡される遅延評価型のどちらかを定義時に選択することができる。

図6はJ. S. Bachのプレリュードをマクロを使って記述した例である。冒頭で1小節分の音型パターンを**pattern**という名のマクロとして定義している。定義中に現れる**$1** などは引数への参照を意味する。引数の種類としては遅延評価型を用いている。このように定義を行うことで、以後は音名を引数としてこのマクロを呼び出すだけで1小節分の記述が完了するようになる。なお、この例では使用されていないが、引数として渡すことのできるのは音名だけでなく、任意のトークン列（例えば、コード、修飾子つきの音符コマンド、フレーズの一部など）を渡すことができる。

マクロ名は各スレッドごとに存在するマクロ辞書

```
defthread(melody_part, chord_part)

scale1 = #(g4, a4,  b4,  c5, d5,  e5,  g5, a5)
scale2 = #(g4, a4,  bb4, c5, c#5, d#5, f5, g4)
scale3 = #(g4, a4,  c5,  d5, e5,  f5,  g5, a5)
scale4 = #(g4, ab4, bb4, b4, c#5, eb5, f5, g5)

rhythms = #( #(q), #(q/3, q/3, q/3),
             #(2q/3, q/3), #(i,i) )

def(rand_phrase) {
  foreach($i, rhythms[irand(#rhythms)]) {
    note(n = $1[frand(8)+1]  l = $i)
  }
}

repeat(16) {
  melody_part {
    repeat(4) { rand_phrase(scale1) }
    repeat(4) { rand_phrase(scale2) }
    repeat(4) { rand_phrase(scale3) }
    repeat(4) { rand_phrase(scale4) }
  }
  chord_part {
    w
    [c3 e3 {h b3 b3}]
    [a2 g3 {h c#4 c#4}]
    [d3 f3 {h c4 c4}]
    [g2 f3 {h b3 b3}]
  }
}
```

図 7: アルゴリズムによる楽曲合成の例

によって管理される。あるスレッドにおいて定義されたマクロは、そのスレッド及びその子孫スレッドから“見える”。このようにマクロ名の通用範囲を限定することによって、ユーザの意図しないマクロ名の衝突を避けている。マクロにはスレッド単位で管理されるものの他に、通常のプログラム言語におけるローカル変数のような、ローカルマクロと呼ばれるあるマクロの展開中だけ有効なマクロが存在する。

5 アルゴリズムによる楽曲合成

PMML は、前節で述べた制御構造とマクロに加えて、汎用インタプリタ言語としての様々な機能を持っている。23 種類の C 言語に類似した演算子を持ち、また、文字列や可変長配列などの構造データ型をサポートしている。更に、乱数生成などの種々のライブラリ関数が提供されている。これらの機能を駆使することによって、アルゴリズムに基づいた楽曲合成が可能である。

図 7は音高およびリズムパターンを乱数によって決定する例である。旋律パート（melody_part）と和音パート（chord_part）という 2 つのパートが存在し、このうち和音パートは C, A7, Dm7, G7 という 4 つのコードを 1 小節に 1 つずつ巡回的に演奏している。これらのコードのそれぞれについてそれに適合する 8 つの音高が scale1 から scale4 という配列に格納されている。旋律パートでは 1/f 乱数を使って 8 つの音高のうちの 1 つをランダムに決定しながら旋律を合成している。この時に使用されるリズムパターンは rhythms という 2 次元配列に 4 種類格納されており、4 分音符ごとにその中からランダムに選ばれている。

6 エフェクタ

PMML におけるエフェクタとは、コマンドによって生成されるイベント（例えば、音符イベントやコントロールチェンジイベント等）に修正を施したり、或はその統計情報を得るために利用されるソフトウェアモジュールのことである。これによって、曲の任意の部分に対して、例えばピッチの変換、時間軸の変換、和音のアルペジオ化、アクセントの自動付けなどの処理を適用することが可能である。エフェクタは概念的には Formula [2] の auxiliary process に類似しているが、ユーザがその動作を自由に定義できるため格段に用途が広い。

エフェクタは、イベントの各々に対してアクションと呼ばれる手続きを呼び出してイベント修正等を実現する。エフェクタを使用するためには、まず、各イベント種類毎のアクションの内容をインタプリタに知らせるために、エフェクタ定義を行わなければならない。エフェクタ定義の結果生成される、各イベント種類毎の処理を記述したデータ構造をエフェクタクラスと呼び、その名前をエフェクタクラス名と言う。

エフェクタを実際に稼働させるためには、定義したエフェクタをスレッドにアタッチする必要がある。アタッチを行うとエフェクタインスタンスと呼ばれるデータ構造がインタプリタ内部に作られる。エフェクタインスタンスに対して、必要ならばエフェクタインスタンス名という名前をつけることができる。スレッドにエフェクタをアタッチすると、そのスレッドにおいて生成されたイベントは、エフェクタインスタンス内に存在するイベントバッファに蓄えられるようになる。そしてスレッドが消滅したとき、イベントバッファに蓄えられていた各イベントに対してアクションが呼び出される。

次に示した例では、1 オクターブ上の音を追加するエフェクタを定義し、アタッチしている。defeff

コマンドはエフェクタ定義を行い、ここでは音符イベントに対するアクションを1オクターブ上の音を新たに追加するものと定めている。エフェクタクラス名はoctaverである。このエフェクタをスレッドにアタッチすると、それ以降そのスレッドにおいて生成される音符はすべてオクターブ奏法で演奏されるようになる。

```
defeff(octaver) { // Define an effector
  case(note) {    // Action for note events
    note(n+=12)&  // Add a note an octave higher
  }
}

octaver()         // Attach the effector

C D E             // These notes will be played
                  // in octaves.
```

また、次は音符のベロシティーを乱数によって変動させるエフェクタの例である。変動の振幅はアタッチのときに引数で与える。たとえば、下の例では引数として5を与えているので、80と指定されている最後の3つの音符のベロシティーは、75から85の間でランダムに定められるようになる。

```
defeff(rand_velocity) { // Define an effector
  case(note) {          // Action for note events
    // irand(a,b) generates a random integer
    // between a and b
    v += irand(-$1,$1)  // Modify the velocity
  }
}

rand_velocity(5)    // Attach the effector
v=80                // Set the velocity to 80
C E G               // The velocity of these
                    // notes will be fluctuated.
```

1つのスレッドに対して複数のエフェクタをアタッチすることができる。このとき、エフェクタの適用は最後にアタッチされたものから先に行われる。例えば、octaverとrand_velocityをこの順にアタッチすると、まずベロシティに乱数が加えられてから1オクターブ上の音が追加されるため、オクターブで演奏される2つの音は同一ベロシティになる。しかし逆順でアタッチした場合は、同じ乱数値が連続しない限り、ベロシティは異なる値である。

アタッチされているエフェクタのリストは各スレッドごとに管理されている。新しいスレッドが生成されたとき、それのエフェクタのリストは親スレッドのものが引き継がれる。たとえば次の例の場合、D音に対しては両方のエフェクタが適用され、C音とE音にはrand_velocityだけが適用されることになる。

```
rand_velocity(5)
C {octaver() D} E
```

各スレッドの各エフェクタインスタンスには有効フラグと呼ばれるビットが割り当てられている。有効フラグが0の間は、たとえアタッチされていてもそのエフェクタによるイベント処理は適用されない。有効フラグの操作はdisableおよびenableコマンドによって行う。有効フラグは、次のようにエフェクタを一時的に無効化するために使われる。

```
octaver()
C                       // Played in octaves
{disable(octaver) G F} // Not played in octaves.
E                       // Played in octaves
```

7 仮想コントローラ

MIDIのコントロールチェンジにおけるコントローラ番号は0から127までと定義されているが、PMMLではこれを255までに拡張している。128番以上のコントローラは仮想コントローラと呼ばれる。一部の仮想コントローラはピッチベンドやテンポ指定などの機能が予め割り付けられている。それ以外の仮想コントローラは、エフェクタを利用することによって、ユーザがその意味を自由に定義できる。例えば、下の例では200番の仮想コントローラをエクスクルーシブメッセージを用いたマスタボリュームとして定義している。msvol_handlerエフェクタは200番のコントロールチェンジイベントをエクスクルーシブメッセージのイベントに置き換える働きがあり、これによってマスターボリュームを通常のコントローラと同じように扱うことが可能になる。

```
defeff(msvol_handler, "", ExpandCtrl) {
  case(ctrl(200)) {
    excl(#(0x7f,0x7f,0x04,0x01,0x00,val))
    reject        // Delete the original event
  }
}

msvol_handler()   // Attach the effector

ctrl(200, 100)    // Set the master volume to 100
E D C
ctrl_to(200, 0, 15u, 1)  // Decrescendo using
                         // master volume
```

仮想コントロールチェンジの最大の利点は、3節で述べた連続コントロールチェンジの手法を仮想コントローラに対しても利用できることである。例えば、ピッチベンド、テンポ、或はマスタボリュームを連続的に変化させることを、ctrl_toやctrl_ctoコマンドによって実現できる（上記の例の最後の行を参照）。

8 PMML インタプリタ

本研究において、PMML のソースコードを読み取り標準 MIDI ファイルを出力する PMML インタプリタを開発した。PMML インタプリタは C 言語を使って書かれ、約 17000 行から成る。これと併せて、良く使われるマクロやエフェクタを定義した標準ライブラリや、音源機種ごとに音色名やエクスクルーシブメッセージによる制御をマクロやエフェクタとして定義した機種依存ライブラリ（どちらも PMML 自身で書かれている）の開発も行った。

PMML インタプリタによって標準 MIDI ファイルへ変換するために要する時間は、ユーザが音楽制作の過程でソースコードの編集と演奏を繰り返し実行する上で十分実用的な値である。演奏時間が数分以下の曲は、おおむね 1 秒以内に変換が完了する。なお、交響曲のように規模の大きな曲の場合は、いくつかのファイルに分割することによって、平均的な変換時間を短縮することが可能である。

9 おわりに

本稿では、イベント記述を対象とした演奏指向言語である PMML についてその概要を述べた。PMML は、タイプ量や可読性を重視した基本的な音楽要素の記述、自由曲線に沿った連続的なパラメータの制御、そしてエフェクタと呼ばれるソフトウェアモジュールによる自由度の高いイベント処理の点において従来の音楽記述言語に比べて優れている。

今後は PMML を楽譜清書や波形合成、さらに入力装置からのイベントをリアルタイムに処理するといった用途にも利用できるように現在の言語仕様を拡張したいと考えている。更に、音楽に同期したコンピュータグラフィックスアニメーションの制作に利用できるように、アニメーションに対する指示をソースコード中に埋め込めるようにしたいと考えている。また、PMML に基づいた効率的な音楽制作を支援するプログラミング環境を整備して行く予定である。

謝辞 秋間の補間法についての御助言をいただいた会津大学の林隆史氏に深く感謝する。

参考文献

[1] Hiroshi Akima. A new method of interpolation and smooth curve fitting based on local procedures. *J. ACM*, 17:589–602, 1970.

[2] David P. Anderson and Ron Kuivila. Formula: A programming language for expressive computer music. *IEEE Computer*, 24(7):12–21, 1991.

[3] Roger B. Dannenberg. *The CMU MIDI Toolkit.* Carnegie-Mellon University, 1988.

[4] NEC Corporation. *N88-BASIC(86) Reference Manual*, 1986.

[5] Bill Schottstaedt. Pla: A composer's idea of a language. *Computer Music Journal*, 7(1):11–20, 1983.

[6] Heinrich Taube. Common music: A music composition language in Common Lisp and CLOS. *Computer Music Journal*, 15(2):21–32, 1991.

[7] 西村 憲. *PMML* ユーザーズ・マニュアル. `http://cglsun4.u-aizu.ac.jp/pmml/`, 1997.