2007－MUS－69 (10)
2007／2／23

リアルタイム音楽情報処理のための並行スクリプト言語SAMUELの概要

西村 憲
会津大学 コンピュータ理工学部

本稿では、著者の提案するSAMUELという並行スクリプト言語について、その概要を述べる。SAMUELは、音楽データ入力、アルゴリズミック作曲、音楽情報処理における研究、およびプログラミング教育のために設計されたオブジェクト指向プログラミング言語である。SAMUELは、並行処理の簡便な記述や、表情情報を伴った音符の簡潔な表記などを特徴とする。開発したSAMUELインタプリタは、リアルタイム・スケジューラを内臓し、時間に従ったMIDIシンセサイザ等の制御やインタラクティブ処理を行う。

Overview of the SAMUEL concurrent scripting language for real-time music information processing

Satoshi Nishimura
School of Computer Science and Engineering
University of Aizu

Abstract

This paper introduces a concurrent scripting language named SAMUEL proposed by the author. SAMUEL is an object-oriented programming language designed for music data entry, algorithmic composition, research on music information processing, and programming education. It features concise description of concurrent processing and brief syntax for describing notes and rests with expressive information. The SAMUEL interpreter developed by the author, incorporating a real-time scheduler, is capable of timed control of MIDI synthesizers and interactive processing.

1 はじめに

小規模プログラム向きのインタプリタ型言語は、スクリプト言語と呼ばれる。スクリプト言語は、近年、プロセッサの処理速度が向上したことで、性能面での問題が緩和され、Cのような機械語コンパイル型、あるいはJavaのような中間言語コンパイル型の言語に取って代わって使われるようになってきている。スクリプト言語はソースファイル変更から実行開始までの待ち時間が小さいため、プログラム開発効率が良く、またプログラミング教育にも適している。

コンピュータ音楽分野においても、これまでさまざまなスクリプト言語が開発・利用されてきた[1]。これらには、作曲用言語、MIDI処理言語、楽音合成言語などがある。作曲用言語はさらに、非アルゴリズミックな記述を扱うデータ入力言語と、アルゴリズミックな記述を扱う手続き型作曲言語に分類できる。

音楽用言語全般について言える現在の問題点は2つある。その1つは、上で述べたデータ入力言語、手続き型作曲言語、MIDI処理言語、そして楽音合成言語が、1つの言語に統一されておらず、それぞれ別の文法を持ち、また処理系が異なるということである。たとえば、手続き型作曲言語として優れているCommonMusic [2]は、データ入力言語としては表情付けなどの点で十分ではない。記述の対象となる曲には、アルゴリズミックな要素と非アルゴリズミックな要素を組み合わせたものもあり、さらにインタラ

クティブ性を取り入れたものもある。また、研究用ツールとしてMIDI処理言語を使う場合、解析の対象とする曲のデータをプログラム中に埋め込みたいことも多い。このような状況で、何種類もの言語を駆使しなければならないのは望ましくない。

筆者もPMMLという表情情報を含めて記述できるデータ入力言語を設計、実装している[3]。PMMLでは、マクロを使用することでアルゴリズミックな記述も可能である。しかし、リアルタイムなインタラクティブ処理には対応していない。また、その他の問題点として、インタプリタ型言語とはいえ、記述した曲を演奏するためには、一旦MIDIファイルに変換しなければならない不便さがあった。さらに、マクロ言語特有の分かりにくさを持っていた。

音楽用言語に関するもう1つの問題点は、必ずしもリアルタイム性が十分ではないということである。MIDI処理言語はLISP、Perl、Pythonなどの汎用スクリプト言語を拡張して実装されることが多いが、音楽用途に適するほど十分なリアルタイム性を持った汎用スクリプト言語の処理系が極めて少ない。したがって、自動伴奏システムのように、入力データをリアルタイムに解析して別の演奏を出力したり、テンポに合わせてシンセサイザの制御を行うには困難を伴う。これは既存の汎用スクリプト言語処理系が、リアルタイム処理を意識して設計されたものではなく、例えばガベージコレクションのために一時的に処理が中断されたりするからである。

一方、リアルタイム性を重視した音楽情報処理における研究ツールとしてはMAX [4] やPure-Data [5] などのビジュアル言語がある。これらも、スクリプト言語と同様、コンパイルを必要としないが、ビジュアル言語で複雑な処理を記述するには限界がある。C言語で書いたものを外部オブジェクトとしてリンクできるが、それではプログラム開発に手間がかかってしまう。

筆者は音楽言語におけるこのような問題を解決するため、2003年より、PMMLを基にして、SAMUELと呼ばれるリアルタイム音楽情報処理のための並列スクリプト言語を設計し、そのインタプリタの実装を続けてきた。以下の章では、その概要について述べる。

2 言語の概要

2.1 SAMUELの目的

SAMUELが想定している適用分野には次の3つがある。

- **作曲** アルゴリズミックな記述、非アルゴリズミックな記述、およびそれらの組み合わせによって曲を記述する。
- **MIDI処理** コンピュータ音楽における研究を目的として、これまでMAXやPure-Dataを使って行っていたようなMIDI処理をスクリプト言語でプログラムする。
- **プログラミング教育** スクリプト言語の利点を生かし、音楽を通して、プログラミングの楽しさを教える。

なお、将来的にはSAMUELを楽音合成にも利用できるように拡張したいと考えている。

2.2 基本的な設計方針

上で述べた目的を達成するため、SAMUELを設計する際には以下のことを念頭においた。

1. PMMLのような表情情報を含めた非アルゴリズミック記述が可能であること。ただし、PMMLとの上位互換性には必ずしも拘らない。
2. 汎用言語としても十分に使用可能であること。
3. シェルのような対話的な実行が可能であること。たとえば、コマンドラインから“C D E”と打ち込むと、即座にシンセサイザによってド、レ、ミの順番で音符が演奏されること。
4. MIDI処理用途に十分なリアルタイム性を備えること。必ずしもデッドラインを守る必要はないという意味でソフトリアルタイムの部類に属するが、ジッタ(遅延のばらつき)が知覚されないように十分配慮する必要がある[6]。
5. クロスプラットフォームであること。

SAMUELの言語仕様を策定するにあたっては、当初、PerlやPythonのような既存のスク

リプト言語の文法を、PMMLの記述も扱えるように拡張することを検討した。しかし、もとから存在する構文を残した上で、"C#" あるいは "[C E G]" といった記述を文として認識させるのは困難だと分かった。この代わりに、PMML記述の始まりと終わりを表す区切り記号を導入し、PMML記述をそれらの間に埋め込むという方式も考えたが、これでは上記の 3. で示したような手軽さが失われてしまう。さらに、すでに非リアルタイム用として構築された処理系を、リアルタイム用に改造するのは容易ではない。そこで、新たな言語として言語仕様を一から設計することとした。

2.3 プログラム言語としてのSAMUEL

音楽記述以外についてのSAMUELの言語仕様は、主にPythonを手本として設計されている。ただし、構文はむしろC言語に近く、フリーフォーマットである。変数に型は無いが、宣言は必要である。ラムダ関数やクロージャなどの機能を備え、関数の引数には省略値やキーワード引数が使える。図1にSAMUELプログラムの例を示す。

SAMUELはオブジェクト指向型言語である。オブジェクトは、整数やシンボルなどのごく少数の基本型と、それ以外の参照型に分類され、参照型のオブジェクトについては、クラス毎にクラスオブジェクトが存在する。ユーザは既存のクラスを拡張して新しいクラスを定義でき、多重継承も可能である。情報隠蔽については、C++やJavaにおけるpublicとprivateに相当するものが提供されている。

2.4 並行計算のモデル

SAMUELでは、スレッドを使った並行プログラミングが可能である。扱えるスレッドは、インタプリタ内の仮想マシンに実装されたユーザレベルのスレッドであり、OSが提供するスレッドではない。このため、生成や消滅は非常に高速である。

スレッドの生成や終了待ちは、SAMUEL特有の構文によって簡潔に表現できる。これは音楽を記述する上でも重要な役割を果たしている。たとえば、2つの関数 `fn1` と `fn2` をそれぞれ別のスレッドを使って並行に実行し、それらが終わるまで待機には、

```
[ { fn1() } { fn2() } ]
```

```
/* define a base class for scales */
class Scale {
  var toneTable = nil  // class variable

  /* constructor */
  defmethod $init(rt) {
    /* add private instance variables */
    var ::*root = rt
    var ::*sz = ::toneTable.size()
  }

  /* method for playing a note */
  defmethod playNote(toneNum) {
    note(root + ::toneTable[toneNum % sz]
         + (toneNum / sz) * 12)
  }

  /* method for playing a scale */
  defmethod play() {
    for( var i = 0; i <= sz; i++ ) {
      ::playNote(i)
    }
  }
}

/* define a derived class */
class MajorScale : Scale {
  var toneTable = [0, 2, 4, 5, 7, 9, 11]
}

/* create a new instance */
var sc = MajorScale(C4)
sc.play()  // call the method
```

図1: プログラムの例

と書けばよい。大括弧で始まるこの構文は、そのまま和音や並行パートを記述するためにも使われ、たとえば、"[C E G]" のようにして和音を表記できる。この他にも、関数を別のスレッドを使って実行する "func()&" という構文や、ループ本体を並行に実行する `parfor` 文などが用意されている。この `parfor` を用いれば、たとえばトーン・クラスタを、

```
parfor(var i = C4; i < C5; i++) {
  note(i)
}
```

ように表現することができる。

スレッド間の通信は、CSP [7] のように、通信チャネルを介してデータを伴ったイベントを送ることを基本とする。ただし、チャネルがイベントのバッファリングを行う点において、CSPとは異なる。また、実行効率を考慮し、Occam [8]

のような共有データへの書き込み禁止は実施していない。

時間に合わせた処理を実現するため、通信チャネルによって送られる各イベントにはタイムスタンプが付けられている。このタイムスタンプはイベント処理のデッドラインとして機能し、各スレッドはできるだけタイムスタンプまでにイベントの処理を終えるようにスケジュールされる。また、現時刻よりもどれだけ先回りして処理してよいかを表す "aheadness" というパラメータを各スレッドに持たせ、ジッタの軽減や時刻のロールバックを可能にしている。

3 非アルゴリズミックな音楽記述

図2に示すように、SAMUELは、PMMLに似た音楽記述を、通常の実行文として受け付ける。さらに、幅広いジャンルの音楽の記述に対応するために、PMMLには無い次のような機能が追加されている。

1. 音名を自由に定義できる。たとえば、

   ```
   defpitch Do = 0
   defpitch Re = 2
   ```

   と予め定義しておくと、"C D#" の代わりに "Do Re#" と表記することが可能となる。

2. 修飾子の意味を変更することが可能である。PMMLやSAMUELでは、音符の後に置いて音符のパラメータを変更する記号列を修飾子と呼んでいる (図2における "\" などがこれに相当する)。PMMLでは修飾子の意味は固定であったが、SAMUELでは該当する関数を再定義することにより変更できる。例えば、微分音のライブラリでは、"@" という修飾子を1/4音ピッチを上げるものとして定義している。

3. 時間の指定を、拍だけでなく秒でも指定でき、混在も可能である。拍と秒の相互変換はテンポマップと呼ばれるオブジェクトによって管理される。テンポマップは各スレッドごとに設定できる。

4 アルゴリズミックな音楽の記述

SAMUELには、アルゴリズミックな音楽記述のためのクラスライブラリが用意されている。この中には、音階のクラス、ジャズ理論に基づいたコードのクラス、音楽の変換を行う様々なエフェクタのクラス、MIDIのピッチベンドによって微分音を扱うためのクラス、CommonMusicにおけるpatternのような各種乱数列や繰り返し数列を発生させるシーケンス・クラスなどが含まれている。もちろん、ユーザが新たなクラスライブラリを構築することも可能である。

```
title("Menuet G-major by J. S. Bach")

// Define two voice parts and set the default
// velocity and octave numbers.
newtrack(rh) { v=80 o=4 }
newtrack(lh) { v=60 o=3 }

// time signature & tempo
beginsong("3/4", 160)

[
  rh: { // description of the right-hand part
    seq(i = 1 to 2) {  // construct a loop
      ^D {G A B ^C}\ ^D G G
      ^ E {C D E F#}\  G _ G G
      ^C {^D ^C B A}\ B {^C B A G}\
      if( i == 1 ) {
        F# {G A B G}\  B\\ A\.~~
      } else {
        A {B A G F#}\  G*.
      }
    }
  }

  lh: { // description of the left-hand part
    [ {G* A} B*. ^D*. ]
    B~~ ^C~~ B~~ A~~ G~~ ^D B G ^D {D ^C B A}\
    B~ A G B G ^C~~ B {^C B A G}\
    A~ F# G~ B
    ^C ^D D G~ _G
  }
]
```

図2: 非アルゴリズミックな記述の例
"C", "D" などは音符を表し、"^" と "_" はオクターブの変更を意味する。また、"\"、"*"、"."、および "~" は、音価を指示するための修飾子である。

図3にアルゴリズミックな記述の例を示す。この例では、2つの並行パートが用いられている。第1のパートは、コードのクラスを使って、4つのコードを巡回的に演奏する。第2のパートは、それぞれのコードに合うスケール上の音を、ブラウン・ノイズに従ってランダムに選びながら演奏する。

```
var nseq = DrunkSeq(0,
                    RandomSeq('uniform',-2,2),
                    -20, 20)

var chord_seq = LoopSeq("CM9", "Eb7-5",
                        "Dm9", "G7alt")

var scale_seq = LoopSeq(MajorScale(C4),
                        Lydian7thScale(Eb4),
                        DorianScale(D4),
                        AlteredScale(G4))
while(true) {
  [ {
      l=1h Chord(chord_seq.next()).play()
    }
    {
      var sc = scale_seq.next()
      repeat(8) { sc.note(nseq.next())(1s) }
    }
  ]
}
```

図 3: アルゴリズミックな記述の例

```
var mi = MidiInput()
while(true) {
  var ev = mi.getEvent()
  if( ev is NoteEvent ) {
    ev.t += 1sec
    ev.n = C4 * 2 - ev.n
  }
  putEvent(ev)
}
```

図 4: インタラクティブ処理の例

5 インタラクティブ処理

SAMUEL には、MidiInput と MidiOutput という、MIDI デバイスとの間でイベントをやりとりするための特殊な通信チャネルが用意されている。これを使うことで、インタラクティブな MIDI 処理が可能である。

図 4 に、簡単なインタラクティブ処理の例を示す。このプログラムは、MIDI 入力からイベントを受け取り、ノート・オン/オフに対して、時間を 1 秒遅らせ、さらにノート番号を C4 音を中心として上下反転させて、MIDI 出力へ送っている。

イベント処理のためのモジュールとして定義されているエフェクタ群も、時間反転などのバッファリングを必須とする一部のものを除き、MIDI 入力からのイベントに対するリアルタイム処理に使用することができる。たとえば、図 4 に示した例は、エフェクタのライブラリを使用すると、

```
{ ModifyNotes('t+=1sec')
  Invert(C4)
  MidiInput().read() }
```

のようにより簡単に表現することが可能である。

6 SAMUEL インタプリタ

6.1 実装

SAMUEL インタプリタは C++言語を使用して実装されており、現在のバージョンでは約 34,000 行のソースコードから成り立っている。シェルのように、コンソールからプログラムコードを受け付け、対話的な実行が可能である。実行速度向上のため、インタプリタは、受け取った SAMUEL ソースコードを、内部で一旦、スタックベースの中間言語にコンパイルしてから実行する（ただし、コンパイルが行われていることはユーザからは見えない）。

リアルタイム・ガベージコレクションには、Dijkstra らのアルゴリズム [9] を使用し、処理の中断を実用上問題のないレベルまで小さく抑えている。

現在までにサポートされているプラットフォームは Windows だけであるが、今後は他の OS に広げてゆく予定である。

6.2 ネイティブ・ランゲージ・インタフェース

SAMUEL は、簡素なネイティブ・ランゲージ・インターフェースを備えている。これによって、SAMUEL から C/C++によって書かれた関数を呼び出すことができ、C/C++で書かれた既存のライブラリをリンクしたり、インタプリタでは性能が十分でない場合に高速化を図ることが可能である。

C/C++で書かれた関数は、SAMUEL では Builtin というクラスのオブジェクトとして扱われる。そして、Builtin クラスのコンストラクタでオブジェクトを生成すれば、SAMUEL から呼べるようになる。例えば、lib.dll というライブラリ・ファイルの中の x という引数をもつ_func という関数を、SAMUEL から func という名で呼ぶことができるようにするには、下の文を実行すれば良い。

```
var func = Builtin("func", "lib.dll",
                   "_func", 'x')
```

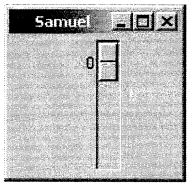

図 5: Tcl/Tk によるスライダ表示

なお、リアルタイム性を保つために、ネイティブ・ランゲージ・インターフェースから呼び出される関数は、呼び出されてから 10 ミリ秒を目安にリターンしなければならないという制約がある。それより実行時間の長い関数の場合には、10 ミリ秒ごとにリターンし、関数が再度呼び出されるのを待ってから続きの処理を行うように、プログラムを変更しなければならない。

6.3 Tcl/Tk との連携

SAMUEL では、OS が提供する GUI 関連の API を直接呼ぶ関数群は提供されていないが、代わりに Tcl/Tk ツールキット [10] を自動的にリンクできるようになっている。これにより、最小の努力で、プラットフォームに依存しない、自由度の高い GUI を提供している。

SAMUEL から Tcl/Tk を使用するには、`tkprintf` という関数を用いる。`tkprintf` は、もし Tcl/Tk をまだリンクしていないのならリンクした後、Tcl/Tk に対してコマンド文字列を送る。一方、Tcl/Tk には、`putevent` というコマンドが追加されており、これにより SAMUEL の通信チャネルに対してイベントを返送することができる。たとえば、

```
var tc = TkEventChannel()
tkprintf("scale .s -command \"putevent %s\"",
        tc.getChannelString())
tkprintf("pack .s")
```

を SAMUEL から実行すると、図 5 のようなスライダが現れ、それを動かす都度、`tc` という通信チャネルにイベントが送られる。`getChannelString` は通信チャネルのハンドル文字列を取得するためのメソッドである。

7 おわりに

本稿では、筆者の提案する SAMUEL という並行スクリプト言語について、その概要を述べた。今後は、言語仕様の詳細や、スレッドスケジューリングのアルゴリズム、および実装技術について逐次公表して行く予定である。開発したインタプリタのソースコードは、現在のところ非公開であるが、今後はフリーウェアとして公開すること検討している。

参考文献

[1] Curtis Roads. *The Computer Music Tutorial*, chapter 17. Addison-Wesley, 1996. (邦訳:コンピュータ音楽 歴史・テクノロジー・アート, 東京電機大学出版局).

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

[3] Satoshi Nishimura. PMML: A music description language supporting algorithmic representation of musical expression. In *Proc. of the 1998 International Computer Music Conference*, pages 171–174, 1998.

[4] Miller Puckette. The pathcer. In *Proc. of the 1988 International Computer Music Conference*, pages 420–429, 1988.

[5] Miller Puckette. Pure data. In *Proc. of the 1996 International Computer Music Conference*, pages 269–272, 1996.

[6] Roger B. Dannenberg and David H. Jameson. Real-time issues in computer music. In *Proc. of the IEEE Real-Time Systems Symposium*, pages 258–260, 1993.

[7] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[8] Alan Burns. *Programming in Occam 2.* Addison-Wesley, 1988.

[9] Edsger W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM*, 21(11):966–975, 1978.

[10] John K. Ousterhout. *Tcl and the Tk Toolkit.* Addison-Wesley, 1994.