Using stochastic petri nets for real-time nth-order stochastic composition

Douglas A. Lyon
Fairfield University, dlyon@fairfield.edu

Douglas Lyon
Computer Science and Engineering Department
Dana Building
University of Bridgeport
Bridgeport, Connecticut 06601, USA
Lyon@cse.bridgeport.edu

Using Stochastic Petri Nets for Real-Time Nth-order Stochastic Composition

This article presents a technique for using stochastic Petri nets for real-time realization of *N*th-order stochastic compositions (a Markov process). The algorithm makes use of a data structure known as a stochastic Petri table. This table is compact and suitable for interactive performance on small computers. We also show how the inherently concurrent nature of Petri nets can be used to implement real-time MIDI processing. Since readers may be unfamiliar with Petri nets, we present a brief introduction to the basic ideas behind the Petri net and compare it with the finite-state machine.

Background

Using a portable computer for real-time composition has a number of advantages over off-line composition techniques (Alles 1977). Real-time composition provides immediate feedback to the composer that can improve productivity. In addition, it is useful in a performance environment.

The use of Markov chains for computer-assisted composition is not new; Stephen Schwanauer and David Levitt's book, *Machine Models of Music*, describe the composition, in 1955, of the *Illiac Suite* by Lejaren Hiller and Leonard Isaacson (Schwanauer and Levitt 1993).

Attempts to realize Markov chain performances in real time are not new either, though the early work here (e.g., O'Haver 1978) only used first-order processes. As far as we know, higher-order Markov chain composition in real time has not been reported in the literature.

A stochastic Petri table is a data structure that is shown to enable the computation of higher-order, real-time Markov processes. In the system described here, we encode a melody using pitch class, and ignore tempo and timbre information to simplify the representation. The computation of the stochastic Petri table is an off-line process. The stochastic Petri table is compact (linear in the number of arcs), and enables real-time Markov chain computation. Our brute-force approach to computing the Petri table is easy to implement and slow to compute. A harder to implement—but faster—approach has been suggested, but remains untried. After the computation of the stochastic Petri table, the program writes it to a file to be read during start-up by the interactive performance program. Thus, this approach allows us to perform a precomputation phase that permits fast execution of the real-time component.

We are motivated to take the stochastic Petri net approach because we have seen other approaches in the literature that do not give real-time performance, and use more memory than is practical on portable computers (Moore 1990). A further motivation for the use of Petri nets is the concurrent nature of user interfaces and music performance. It is often the case, for example, that a user will generate interrupts during a performance. As shown below, these interrupts cannot be handled with a finite-state machine, but can be handled by Petri nets.

Limits of the Finite-state Machine Model

In this section we define the Turing machine, the finite-state machine, and identify the limitations of the finite-state machine model. We use these limitations to motivate our use of the Petri-net model that is discussed next.

A Turing machine is an imaginary computing device that consists of a control unit (which may assume one state at a time), a tape (which can store a

Computer Music Journal, 19:4, pp. 13–22, Winter 1995
© 1995 Massachusetts Institute of Technology.

symbol), and a read-write head (which moves relative to the tape and can relay information between the control unit and the tape).

A finite-state machine is a deterministic device with a fixed number of states. A special case of the Turing machine, the finite-state machine is also known as a finite automaton. The finite-state machine consists of a Turing machine with a single input tape and a read-only head (Ralston 1983). The output and next-state of a finite-state machine are a function of the machine's present state and inputs (Katz 1994).

Finite-state machines are often depicted by state diagrams (also called transition diagrams), which are directed graphs that show a finite-state machine's input and output. A state diagram that represents a gum-vending machine is shown in Figure 1. The circles in Figure 1 represent states; each is labeled with the amount of money that has been put into the machine to get it into that state. The arrows represent transitions; each is labeled with the amount of money that must be put into the machine to cause that transition.

One problem, though, is that the finite-state machine model does not handle interrupts well. From a programmer's view, a sub-routine is serviced after the state of the machine is pushed onto a stack. The finite-state machine does not provide a stack, nor does it describe how an interrupt should be serviced, or to where it should return.

A further limit of the finite-state machine model is that it cannot handle multiple asynchronous processes—it cannot "be" in multiple states at one time. This is really a result of its inability to allow for the synchronization of parallel activities. One way to perform this synchronization is through the use of token passing.

Petri nets (discussed in the next section) can model this concurrency and handle and recover from interrupts, two features that are notably lacking in finite-state machines.

Petri Nets

A Petri net is a bipartite, directed graph that uses tokens to enable computations. The graph is bipartite because it uses two kinds of nodes called *places* and *transitions*. The graph is directed because all connections in the graph consist of *directed arcs* that lead from places to transitions or from transitions to places. Data entities known as *tokens* travel along the arcs and enable computation (Peterson 1977).

In the Petri net, places symbolize conditions and transitions represent computations. In addition, every transition is connected to *input* and *output places*.

The Petri net may be represented graphically using a Petri net diagram, or textually using a Petri net table. The primitives of the Petri net diagram are shown in Figure 2. The diagram is better suited for human communication, while the table is better suited for machine communication. Figure 3 shows an example of a Petri net for a gum machine represented by both a Petri net diagram (Figure 3a) and a Petri net table (Figure 3b).

Figure 1. A finite-state machine diagram for a gum-vending machine. The circles represent states; each is labeled with the amount of money that has been put into the machine to get it into that state. The arrows represent transitions; each is labeled with the amount of money that must be put into the machine to cause that transition. The price of gum from this machine is 15¢. The Petri net can represent any finite-state machine, but the opposite is not true.

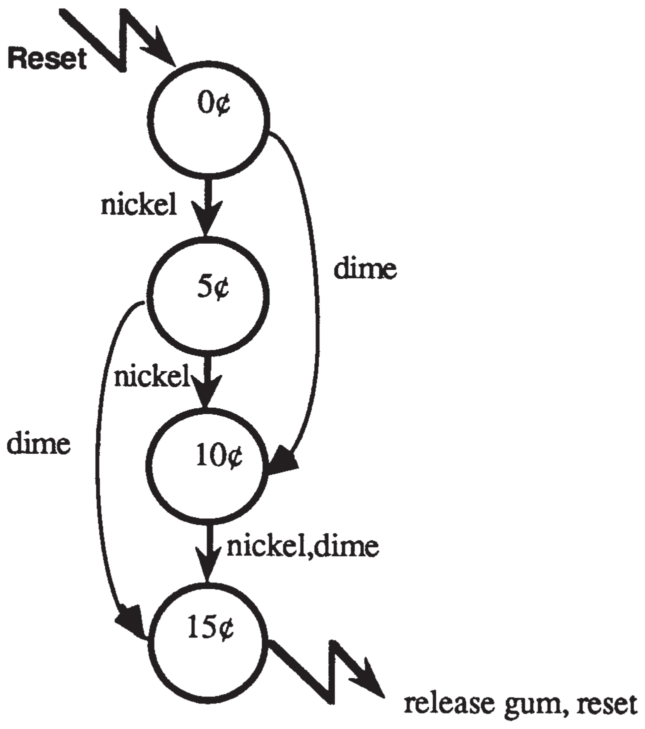
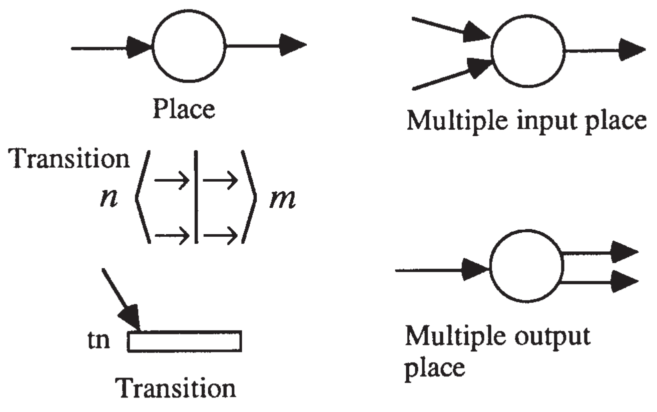


Figure 2. Petri net primitives. The Petri net is a bipartite graph, because there are two types of nodes: places and transitions. It is a directed graph, because all arcs connect from a place to a transition, or from a transition to a place. In this article, the transition is sometimes represented by a straight line, and other times by a hollow rectangle.

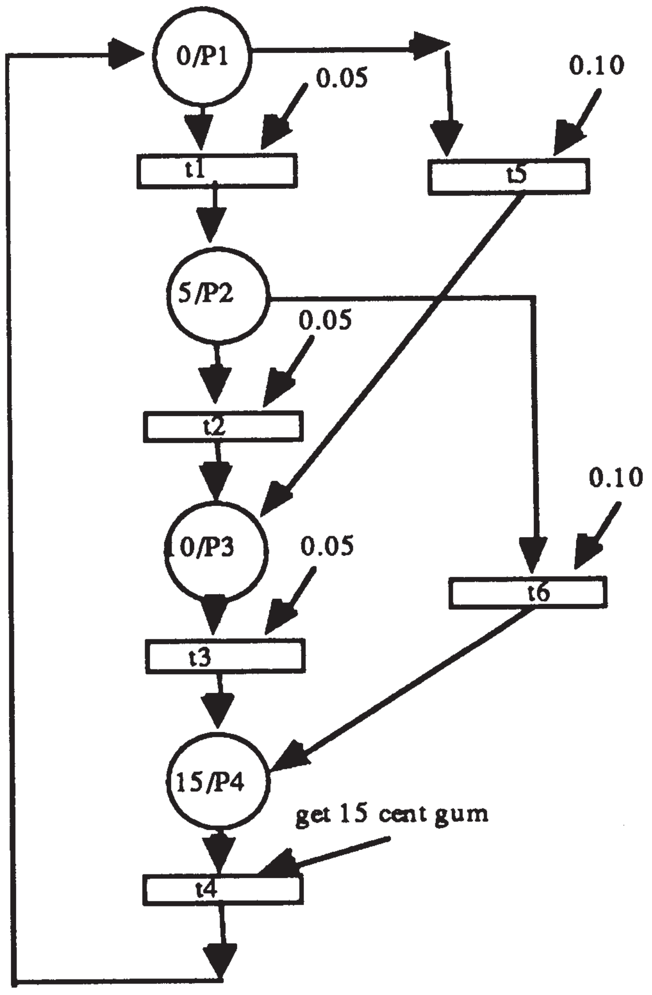
Figure 3. Petri diagram (a) and table (b) for a gum machine. The price of gum is 15¢ and no change is given. The stochastic Petri table shows the present place, next place, probability of transition, token name, and action caused. The probability of leaving state zero is 0.15. The probability of remaining in state zero is 1 − 0.15 = 0.85. This example could be implemented as a Markov chain.



The Petri net in Figure 3 represents the same system that was modeled with a finite-state machine in Figure 1. The finite-state machine cannot model interrupts, and these are present in the example of a real-time MIDI delay system, shown in Figure 4. The real-time MIDI delay system is designed to act like a 3-sec echo box with no decay and some maximum number of echoes for every MIDI event. This has advantages over traditional approaches to achieving a 3-sec delay: there is no noise or decay in the repeated event, and the number of echoes is a parameter that may be set by the performer.

The interrupt generated by the user, shown in Figure 4 by the *Mouse_button* token, is typical of user-generated events. User input occurs concurrently with the execution of the main body of the program, and is asynchronous with respect to the execution of the code.

The Petri net in Figure 4 is depicted as a Petri table in Figure 5. A fragment of the Pascal language source code that implements that Petri table is given in Figure 6. The system from which this excerpt is taken was developed using Symantec Think Pascal (Symantec 1991) on an Apple Macintosh computer. The code was designed to be executed in a sequential fashion, so it does not enable the handling of true interrupts. Instead, the events are queued by the operating system and de-queued by the *repeat [. . .] until button* main-event loop of the program. For the case of parallel Pascal, a more direct translation from the Petri net to the code is possible. The advantage of this approach over the



(a)

Name	*Place*	*Enabling Tokens*	*Transitions*	*Actions*	*Next Place*
0¢	p1	0.05	t1	-	p2
5¢	p2	0.05	t2	-	p3
10¢	p3	0.05	t3	-	p4
15¢	p4	get gum	t4	get gum	p1
0¢	p1	0.10	t5	-	p3
5¢	p2	0.10	t6	-	p4

(b)

finite-state diagram is the explicit use of interrupts in the highest level of design.

Markov Chains

In this section we present the Markov chain abstraction, and show how it can be implemented

Figure 4. A Petri net for a real-time 3-sec MIDI delay. From any place in the Petri net (P1...P 7), an interrupt in the main-event loop may be generated by the user's input of a mouse click. Interrupts are hard to implement with finite-state machines.

Figure 5. A Petri table for a 3-sec delay. This table represents the same Petri net shown by the Petri diagram of Figure 4. The ● in the last row, second column of the Petri table indicates that the mouse_button *token will cause a jump from any place in the Petri net.*

Figure 6. Pascal code for the 3-sec MIDI delay net.

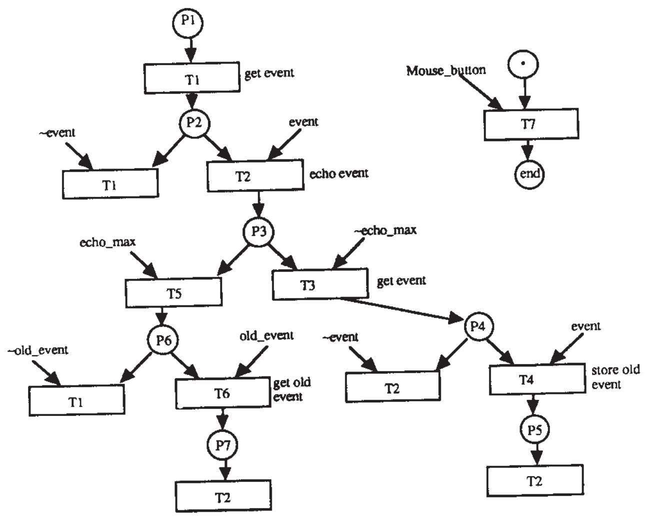


Figure 4

Name	Place	Enabling Tokens	Transitions	Actions	Next Place
start	p1	-	t1	get event	p2
	p2	event	t2	echo event	p3
	p2	~event	t1	get event	p2
	p3	echo_max	t5	check if q is empty	p6
	p3	~echo_max	t3	get event	p4
	p4	event	t4	store old event	p5
	p4	~event	t2	echo event	p3
	p5	-	t2	echo event	p1
	p6	~old_event	t1	get event	p1
	p6	old_event	t6	get old event	p7
	p7	-	t2	echo event	p3
	•	mouse_button	t7	quit midi	end

Figure 5

```
begin { of main}
 initialize_program;
 transition := 1;
 repeat
  case transition of
   1:  {scan for an event }
    begin
     if midi_get(event) then {p1}
      transition := 2 {got an event}
     else
      transition := 1;
    end; {case 1}
   2: {output note }
    begin
     echo_event(event);
     if event.number_of_times_put < max_echos then
      transition := 3
     else
      transition := 5;
                  {ok, we echoed enough times, check the queue for old events}
    end; {case 2}
   3:  {update and start again if any new events occur}
    begin
     if midi_get(event2) then
      transition := 4 {a new event occured while we were echoing}
     else
      transition := 2; {keep echoing, no new inputs}
    end; {case 3}
   4:
    begin
    (event); {keep  old event for later}
     w-iteln('stashing old event');
     copy_event(event2, event);
     transition := 2;
    end; {case 4}
   5:
    begin
     if q_empty then
      transition := 1 {scan for fresh events}
     else
      transition := 6;
    end; {case 5}
   6:
    begin {queue is not empty so get stashed event and echo it}
     remove_q(event);
     writeln('removed old event');
     transition := 2;
    end; {case 6}
  end; {case}
 until button;
 QuitMidi; {discards memory and removes interrupt handlers}
end.
```

using transition tables. A Markov chain is a non-deterministic finite-state machine; it can be represented by assigning probabilities to the transitions in the finite-state machine. In addition, the sum of all the probabilities leaving any Markov state must be equal to one.

A Markov random process is classified as being *continuous-valued* or *discrete-valued*. For the purpose of selecting pitches from a scale (finite set), we use the *discrete-valued Markov random process* (DVMRP or Markov chain). A *Markov chain* is a DVMRP with a countable or finite set of states.

The Markov chain satisfies the conditional probability mass function expression

$$P_x(x_n|x_{n-1},x_{n-2},\ldots,x_1;t_n,\ldots,t_1) = P_x(x_n|x_{n-1};t_n,t_{n-1})$$

for all $x_1,\ldots,x_n$ and for all $t_1 < \ldots < t_n$ and for all $n > 0$. The value of the random variable X at time t will determine the conditional probabilities for the future process values. The process values are called the *process state*, and the conditional probabilities are called the *transition probabilities* between the states. By observing many events, a program computes the probability that X will have a specific value x at a particular time t. This is denoted $P_x(x|t)$. A Markov process is *stationary* if the probabilities are static. For our system, we assume that the Markov process is stationary because we perform off-line analysis.

The transition table of probabilities uses

$$P\{ X_n = i|X_{n-1} = j\} = p_{ij}.$$

The resulting two-dimensional table represents the transition probabilities in a first-order Markov chain, as discussed by Charles Dodge and Thomas Jerse in their book *Computer Music* (1985). Like F. Richard Moore, these authors describe an $N + 1$-dimensioned table to represent an Nth order Markov process. The brute-force approach to the composition using Markov chains usually centers upon the creation of such a transition table.

This table of arc-transition probabilities is usually sparse, and so it requires a program to perform access and computation with a large, higher-dimensioned matrix containing many zero elements. The probabilities assigned to the arcs may be arrived at by one of several methods. We use a technique of statistical analysis of an existing piece of music and have found this method described in the literature (Moore 1990).

The *order* of a Markov process indicates the amount of event memory that the process has. For example, a zeroth-order Markov process has no event memory. A first-order Markov process takes into account a single "historical" event, and an Nth-order Markov process takes into account the last N events.

To perform the analysis of a melody, we create a list of the pitch classes of the notes. For example, in the main theme of Louis Bonfi's *Black Orpheus*, the notes are:

E, C, B, A, A, G#, B, E, E, C, B, A, A, G, B, E, E, F, G, A, D, D, D, E, F, G, C, C, C, D, E, F, B, B, C, D, E, E, C, B, A, A, G#, B, E, E, A#, A, G, G, F, E, A, D, D, E, F, G, C, C, D, E, A, G#, E, E, G#, B, A, E, A, A, B, C, D, C, B, A, B, C, D, C, B, A, B, C, D, C, B, A, G.

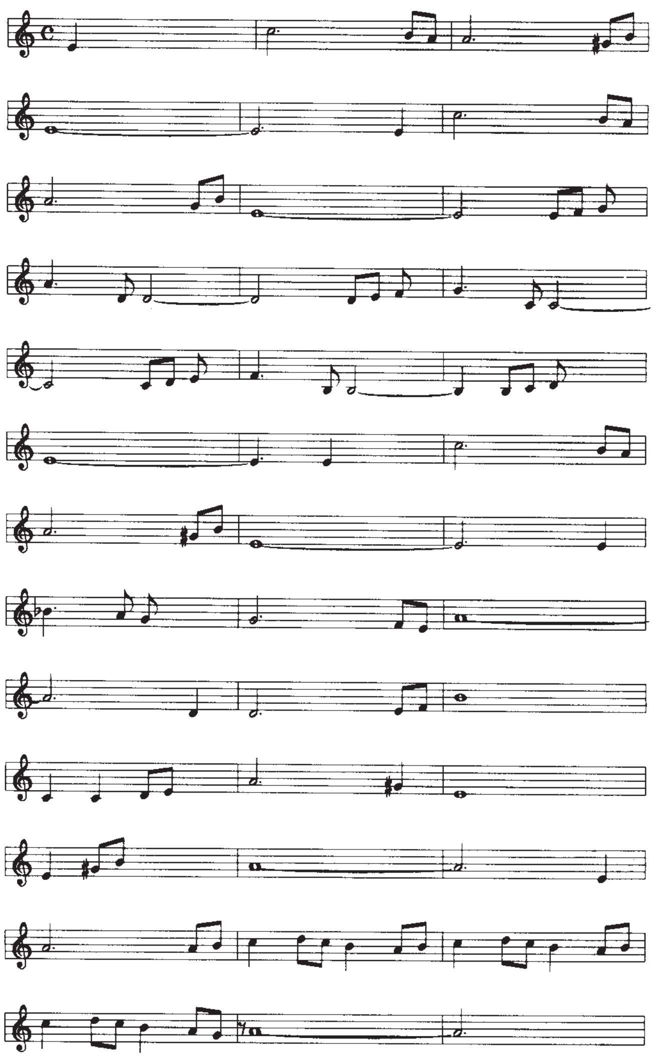
Converting into a pitch class requires that each note be assigned a number, for example:

[A A# B C C# D D# E F F# G G#] = [1 2 3 4 5 6 7 8 9 10 11 12].

The *Black Orpheus* theme then becomes:

8, 4, 3, 1, 1, 12, 3, 8, 8, 4, 3, 1, 1, 11, 3, 8, 8, 9, 11, 1, 6, 6, 6, 8, 9, 11, 4, 4, 4, 6, 8, 9, 3, 3, 4, 6, 8, 8, 4, 3, 1, 1, 12, 3, 8, 8, 2, 1, 11, 11, 9, 8, 1, 6, 6, 8, 9, 11, 4, 4, 6, 8, 1, 12, 8, 8, 12, 3, 1, 8, 1, 1, 3, 4, 6, 4, 3, 1, 3, 4, 6, 4, 3, 1, 3, 4, 6, 4, 3, 1, 11.

The technique of converting a melody into its corresponding pitch class is not new (Winsor 1987).



These notes are hand-written into a file called `notes`, and then read into an array of integers called *note_array* by the program.

Assume that a note's occurrence is an independent, random event. We have written a procedure that compiles a table to record the frequency of occurrence into an array called the *pmf_array[i]*. Here, PMF is an abbreviation for the probability mass function. This is a statistical record of the frequency of occurrence of each note in the melody. It is treated as a discrete probability distribution function so that

Figure 7. Code for computation of the Probability Mass Function (PMF).

```
Compute the probability mass function array, pmf_array
random_pick = random(0,1)
sum = 0
i = 1
repeat
      sum = sum + pmf_array[i]
      i = i + 1
until sum >= random_pick
play_note(i)
```

$$1 = \sum_{i=1}^{N} pmf\_array[i]$$

must be true.

We use the probability mass function array to bias our choice of a note by picking a random number, *random_pick* from the range of 0 to 1. We then compute the cumulative mass function by summing the elements of the probability mass function array until they exceed the value of *random_pick*. This is shown in the pseudocode given in Figure 7.

A first-order Markov process requires that the transitions be used to compute a transition table that records the frequency of occurrence of each note. Each element in the transition table represents the probability of that note being played. We compute the elements in the transition table by summing the number of transitions for each row and dividing each element in the row by that sum. This "normalizes" the PMFs so that each row will add up to one, i.e.,

$$\sum_{i=1}^{N} P_{ij} = 1 \ \forall_j \in [1 \ldots N].$$

The transition table for a first-order Markov process description of *Black Orpheus* appears in Figure 8. It is possible to transform this transition table into a Markov diagram, but the results are cluttered. The advantage of using the transition table is that it provides a compact and convenient form for representing and programming *first-order* stochastic processes. Using this technique, an Nth-order Markov Process requires an N + 1-dimensional transition table.

Suppose that we wish to compute Markov process probability tables from order 0 to 9, and store the results in the computer's memory. In general, the number of cells needed in all Nth-order stochastic processes from 0 to 9 involving p pitch classes has

Figure 8. A transition table for a first-order Markov process. In general, as the order of the process increases, the sparsity of the matrix increases. Using this technique, an Nth-order Markov process requires an N+1-dimensional transition table.

	A	A#	B	C	D	E	F	G	G#
A	4/19		3/19		2/19	1/19		6/19	3/19
A#	1								
B	7/15		1/15	4/15		3/15			
C			6/15	3/15	6/15				
D				3/11	3/11	5/11			
E	4/19	1/19		3/19		5/19	4/19	1/19	1/19
F			1/5			1/5		3/5	
G	1/5		1/5	2/5				1/5	
G#			3/4			1/4			

$$f(p,N) = \sum_{i=0}^{N} p^{i+1} = \frac{p^2}{p-1}[p^N - 1] \qquad (1)$$

elements when all the Nth-order processes are stored in N + 1-dimensional matrices. For twelve pitch classes and ninth-order processes, this gives a value of $f(12, 9) \approx 6.75 \times 10^{10}$, and any attempt to reduce this number may be foiled by the introduction of pathologic data created by an advisory. For example, a large number of random numbers (much larger than the number of elements in the matrices) will eventually fill all the elements in the matrices with non-zero values. This is sometimes referred to as a zeroth-order stochastic process. Nevertheless, it is practical, for low values of N, to perform the computation as described above. For example, Figure 9 shows the code needed to compute the first-order Markov chain for *Black Orpheus*.

Using this approach, we can implement a second-order Markov process using the data structure shown in Figure 10.

We find, using equation (1), that a ninth-order stochastic process over a twelve-tone system must make use of approximately 6.75×10^{10} elements. To make matters worse, using a matrix approach requires that we iterate over zero-valued elements

Figure 9. The pseudocode for computing a first-order stochastic composition.

Figure 10. The data structure for a second-order Markov array.

Figure 11. A fragment of a first-order Petri net table. Removal of the zero elements in the Markov table reduces the branch-computation time. The number of rows is equal to the number of transitions.

Figure 12. The file format for the stochastic Petri table. The columns appear in the order `place, probability, next-place`.

```
procedure compute_1st-order
begin
    subtract the order number to keep
    the window from exceeding the number
    of notes -- number_of_notes - order,
    note, the number_of_notes > order
    with notes do
        for each (note1 and note2) in
            (note_array[i] and
             note_array[i+1])
        do
            increment the
                number_of_2nd_order_event
                    [note1,note2]
            normalize the probabilities
            in the 1st-order matrix
end {compute first-order}


procedure play_first_order_
    stochastic_note
begin
    pick = random(0,1)
    use the current row to perform a
        biased choice
    play the choice and store the
        next element
end {play_first_order_stochastic_note}
```

Figure 10

Name	Place	Enabling Tokens	Probability	Next Place
A	p1	Ta	0.21052632	p1
A	p1	Tb	0.15789474	p3
A	p1	Td	0.10526316	p4
A	p1	Te	0.05263158	p5
A	p1	Tg	0.31578947	p6
A	p1	Tg#	0.15789474	p7
A#	p2	Ta	1.00	p1
B	p3	Ta	0.46666667	p2
B	p3	Tb	0.06666667	p3
B	p3	Tc	0.26666667	p4

Figure 11

1	0.21052632	1
1	0.15789474	3
1	0.10526316	4
1	0.05263158	5
1	0.31578947	6
1	0.15789474	7
2	1	1
3	0.46666667	2

Figure 12

when computing the cumulative mass function. One objective—a prerequisite for the interactive realization of Markov processes on small computers—is to minimize the time it takes to compute a branch between nodes in the Markov chain.

Algorithms are known (Press et al. 1992) that implement sparse matrices, which require space proportional to the number of elements. These sparse-matrix approaches reduce the amount of space needed, but cannot address the issue of execution-time reduction. Since many visited states have zero probability (using the matrix approach), the time that the program takes to compute a branch cannot be predicted. We have found that this creates uneven playback of the Markov chain in real-time performance. This is a primary motivation for our approach, which has space requirements that are linear in the number of Markov states.

Stochastic Petri Nets

The Petri net approach removes the zero elements in the transition table. The Petri table for the first-order stochastic process shown above appears in Figure 11. The program shown in Figure 13 implements a second-order Markov process using a Petri net. *Transition probabilities* are defined as the conditional probabilities for moving between pairs of states.

To implement the Petri table and take full advantage of the fact that the number of elements in the table is linear with respect to the number of transitions in the stochastic process, we need to write a program that can read Petri tables directly. First we establish a Petri table file format for a Markov chain that is stored as `[place, probability, next-place]`. This file format is illustrated in Figure 12.

The Pascal data structure used to store the Petri net, with the next states and their probabilities, is defined in Figure 13. The pseudocode in Figure 14 plays a row in the Petri table. Here, CMF stands for the cumulative probability mass function (computed by summing the probabilities of each of the transition arcs). To better understand the program, consider the Petri table shown in Figure 12. To play one row, we first pick a pseudorandom number from 0 to 1, inclusive. We then step from one row of the Petri table to the next until the cumulative mass function exceeds our probability pick. When

Figure 13. Declaration of the stochastic Petri table data type.

```
type  {Petri-markov data types}
{ Data type used for the rows of the table in Figure 12 }
  petri_row_record = record
    place: integer;
    probability: real;
    next_place: integer;
   end;     {petri_row_record}

  row_array_type = array[1..35] of petri_row_record;

{ Data type used for the Petri table }
  petri_type = record
    row_array: row_array_type;
    number_of_rows: integer;
    current_place: integer;
   end;     {petri_array_type}

{ Declare one table }
var
  petri: petri_type;
```

Figure 14. Code to play notes using the Petri table.

```
procedure play_petri_row (var petri: petri_type);
{play one row in the petri-table implementation
       of the first order Markov chain}
var
   i: integer;
   place_name: integer;
   cmf: real;
   prob_pick: real; {a number between 0 and 1}
   while_loop_not_done: boolean;

begin
{compute the cumulative probability mass function}
  cmf = 0;
  prob_pick = random(0,1)
  { Until we exit }
  while_loop_not_done := true;
  with petri do
   begin
    i := current_place;
    place_name := row_array[i].place;
    with row_array[i] do
     begin
      while (place_name = place)
            and (cdf < prob_pick) do
       begin
       { Sum up the probabilities into the CMF }
       cmf := cmf + probability;
       i := i + 1; {move to next row}
       end; {while place_name}
      { Play the chosen note }
      make_tone(scale_array[i], tone_time);
      current_place := next_place;
     end; {with row_array[i]}
   end; {with petri}
 end; {play_petri_row}
```

this occurs, we take the transition to the place that corresponds to the row at which we stopped.

In case the cumulative mass function does not add up to one (due to round-off error), we provide a compound conditional test in the `while` loop, `((place_name = place) and (cmf < prob_pick))`, that keeps us within the Petri table row.

The Petri table is a faster method for realizing Markov chains than the Markov table, because of the elimination of the zero elements. It would be even faster to store the cumulative mass function, rather than the probability mass function. This saves a floating-point addition as the program iterates over each element in the Petri table. The transition matrix requires 144 real numbers for a first-order Markov process. The Petri table needs only 66 integers and 33 reals. Assuming that an integer is 2 bytes long and that a real is 4 bytes, there are 144×4 bytes per real, or 576 bytes used for the Markov transition table and $66 \times 2 + 33 \times 4 = 264$ bytes for the Petri table. This saving grows as an exponential function of the order of the Markov process.

Suppose we use the *Black Orpheus* example to compute a second-order Markov chain. To speed execution and eliminate the sparse matrices, we use a Petri net. A partial Petri net implementation for this Markov chain is shown in Figure 15. Here, we note that each of places has a transition probability that sums to one. Let R be the row number of the Petri table, and $N1$, $N2$, and $N3$ represent note 1, note 2, and note 3. $Pijk$ is the probability that note k will occur given the occurrence of notes i and j. A partial Petri table for this is shown in Figure 16.

We have found that, for the *Black Orpheus* example, there are 378 elements in a forth-order Petri table. If a forth-order transition table were used, it would need $11^5 = 161{,}051$ elements, about 426 times more storage. Such improvements must be viewed with cautious optimism—more experimentation is needed.

To play such a table, a program must jump to a row, given two notes. A procedure picks a uniformly distributed random number that varies from 0 to 1 (called r, created using a linear congruential random number generator with a long period) (L'Ecuyer, Blouin, and Couture 1993). The procedure then sums the probabilities to form the cumulative mass function. When the CMF exceeds r, the value for the next state, k, is obtained. For example, sup-

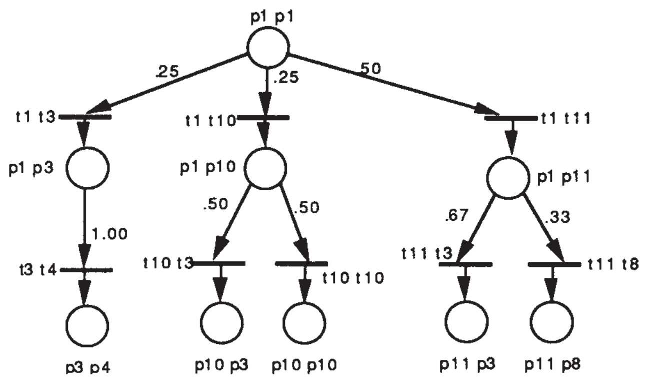


Figure 15

Figure 15. A partial stochastic Petri net. The transition probabilities are shown on each arc. They indicate the probability of the occurrence of a token.

R	N1	N2	N3	Pijk
1	1	1	3	0.25
2	1	1	10	0.25
3	1	1	11	0.50
4	1	3	4	1.00
5	1	6	6	1.00
6	1	8	1	1.00
7	1	10	3	0.50
8	1	10	10	0.50
9	1	11	3	0.67
10	1	11	8	0.33
11	2	1	10	1.00
12	3	1	1	0.43
13	3	1	3	0.29
14	3	1	8	0.14
15	3	1	10	0.14
16	3	3	4	1.00
17	3	4	6	1.00
18	3	8	8	1.00

Figure 16

Figure 16. A partial stochastic Petri table. R *is the row number of the table.* N1 *and* N2 *are two notes in the history of the Markov chain.* N3 *is the next note in the chain and* Pijk *is the probability that* N3 *will occur given the occurrence of notes* N1 *and* N2.

pose $r = 0.30$. Starting on row 1 in Figure 16, adding $P_{1,1,3} + P_{1,1,10} + P_{1,1,11} = 0.50 > 0.30$. In Petri net parlance, k is a token that appears with a given probability. Note $N1$ becomes the old $N2$, $N2$ becomes the old $N3$, and $N3$ takes on the value of the `next_place`, or, $N1 \leftarrow N2 \leftarrow N3 \leftarrow next\_place$. For added efficiency, an array (called the *indirection array*) gives us a pointer to the beginning of the list

Figure 17. An indirection array. The first element indicates the next note's pitch class. The second element points to the first row in the Petri table where the pitch class may be found. This keeps the procedure from having to perform a search for the next pitch class.

1	1
2	11
3	12
4	19
5	0
6	24
7	0
8	30
9	42
10	46
11	51

of places that start with note number one. An example indirection array is shown in Figure 17.

We have implemented fifth-order stochastic processes using Petri nets. After that, the 100-note example tends to be deterministic. With the ability to alter the order of stochastic control, in real time, the author has been able to obtain playback at a rate of 180 notes per sec. This rate was measured on an Apple Macintosh PowerBook 165 (which uses a Motorola MC68030 processor running at 33 MHz with no math coprocessor) with sequentially changing orders of stochastic control (3, 4, and 5) over a 10-sec interval. Since fifth-order stochastic control is almost deterministic (for our example) it takes very little time to compute the branch (on average).

Conclusion

We have shown the stochastic Petri net paradigm may be used to create an efficient method for computing real-time Markov chains for composition and real-time interaction. We have also used the Petri net paradigm to show how concurrent asynchronous user inputs may be specified in a high-level manner. The use of stochastic Petri nets to perform real-time Markov processes is, as far as we know, novel.

In the future, we could eliminate the computation of the cumulative mass function by storing the cumulative mass function rather than the probability mass function. This would save the cost of one addition during the branching computation. In addition, we think that the pre-computation of the stochastic Petri table could be made faster than the present implementation.

This author believes that the Markov chain method of computation may be useful in the areas of network protocol simulation, parallel computer simulation, animation, and computer-assisted dance. On this topic, as on many others relating to the use of stochastic Petri nets in the arts, much work remains to be done.

The source code and Macintosh-compiled version of the program described in this article are available from the *Computer Music Journal*'s World-Wide Web site in the directory with the uniform resource locator ftp://www-mitpress.mit.edu/pub/Computer-Music-Journal/Code/Lyon.

Acknowledgments

Special thanks go to Julius Dichter, for his careful proofreading. Thanks also go to Stephen Travis Pope and the *Computer Music Journal* referees for their encouragement and suggestions.

This project was made possible, in part, by an Instrumentation Laboratory Improvement grant from the National Science Foundation, DUE–9451520, and by a Larsen Professor grant from the Larsen Fund.

I would also like to thank the people who built the Video Synthesis Laboratory at Rensselaer Polytechnic Institute (RPI). It was through their volunteer efforts that an experimental electronic music facility first appeared at RPI. Major contributions were made by Tom DeWitt, Aaron Heller, Jim Hill, Danny Rosenberg, Chris Shenton, Dean Winkler, and others. I would also like to thank Joel Chadabe for allowing me to use the Electronic Music Studio at the State University of New York at Albany.

Finally, I wish to acknowledge the debt I have to the late Mike Torello, who died while this article was being written. Co-performer/co-composer and friend, Mike's influence and enthusiasm were infectious. He continues to be missed.

References

Alles, H. G. 1977. "A Portable Digital Sound Synthesis System." *Computer Music Journal* 1(4):44–49.

Batish, S. D., and A. Batish. 1989. *Ragopedia Volume 1.* 1316 Mission St., Santa Cruz, California 96060: Batish Publications.

Dodge, C., and T. Jerse. 1985. *Computer Music.* New York: Schimer.

Katz, R. 1994. *Contemporary Logic Design.* New York: Benjamin/Cummings.

L'Ecuyer, P., F. Blouin, and R. Couture. 1993. "A Search for Good Multiple Recursive Random Number Generators." *ACM Transactions on Modeling and Computer Simulation* 3(2):87–98.

Moore, F. R. 1990. *Elements of Computer Music.* Englewood Cliffs, New Jersey: Prentice-Hall.

O'Haver, T. 1978. "More Music for the 6502." *Byte* 3(6):140–141.

Peterson, J. 1977. "Petri Nets." *ACM Computing Surveys* 9(3):223–252.

Press, W., et al. 1992. *Numerical Recipes in C.* Cambridge, UK: Cambridge University Press.

Ralston, A. 1983. *Encyclopedia of Computer Science and Engineering*, 2nd. ed., New York: Van Nostrand Reinhold.

Schwanauer, S., and D. Levitt. 1993. *Machine Models of Music.* Cambridge, Massachusetts: MIT Press.

Symantec Corporation. 1991. Think *Pascal User Manual.* Cupertino, California: Symantec.

Winsor, P. 1987. *Computer-Assisted Music Composition.* Princeton, New Jersey: Petrocelli.