Technical Report #9602

MODELING USING TIMED PETRI NETS – EVENT–DRIVEN SIMULATION

by

W.M. Zuberek

Department of Computer Science
Memorial University of Newfoundland
St. John's, Canada A1B 3X5

September 1996

Department of Computer Science
Memorial University of Newfoundland
St. John's, Canada A1B 3X5
tel: (709) 737–8627
fax: (709) 737–2009

Copyright © 1996 by W.M. Zuberek.
All rights reserved.

The Natural Sciences and Engineering Research Council of Canada
partially supported this research through Research Grant A8222.

MODELING USING TIMED PETRI NETS – EVENT–DRIVEN SIMULATION

Abstract

A collection of software tools for analysis of timed Petri nets, *TPN–tools*, developed over years of extensions, modifications and redesigns, contains several tools for structural and reachability analysis of net models. As both structural and reachability analyses impose certain restrictions on the class on analyzed nets, a simulation tool, *TPNsim*, has recently been added to the collection. All these tools use the same (internal) representation of nets, so the integration of different tools is quite straightforward.

This report briefly introduces the principle of event–driven simulation, and outlines its implementation in *TPNsim*. Processing of transition firings, event scheduling and conflict resolutions are discussed in greater detail. Several examples are provided to illustrate the use of model simulations.

Acknowledgement

Software tools for analysis of timed Petri net models were developed over many years, using direct and indirect contributions of many colleagues and students. All these contributions, rather difficult to trace and identify because of countless software modifications, redesigns and extensions, are appreciated and gratefully acknowledged.

INTRODUCTION

Petri nets have been proposed as a simple and convenient formalism for modeling systems that exhibit parallel and concurrent activities [Ag79, Pe81, Re85, Mu89]. In order to take also the durations of these activities into account, several types of Petri nets 'with time' have been proposed by assigning 'firing times' to the transitions or places of a net. In timed nets, firing times are associates with transitions, and transition firings are 'real–time' events, i.e., tokens are removed from input places at the beginning of the firing period, and they are deposited to the output places at the end of this period (sometimes this is also called a 'three–phase' firing mechanism). In stochastic (and generalized stochastic) Petri nets [Mo82, AM84], (exponentially distributed) firing times are associated with transitions, but the tokens remain (for the firing time) in their "home" places, and the instatntaneous firings occur at the end of firing times. In time nets [MF76, Aa93] there is an interval associated with a transition, and the (instantaneous) firing must occur within this interval of time.

In timed nets, all firings of enabled transitions are initiated in the same instants of time in which the transitions become enabled (although some enabled transition cannot initiate their firing; for example, all transitions in a free–choice class can be enabled, but only one can fire). If, during the firing period of a transition, the transition becomes enabled again, a new, independent firing can be initiated, which will 'overlap' with the other firing(s). There is no limit on the number of simultaneous firings of the same transition (sometimes this is called 'infinite firing semantics'). Similarly, if a transition is enabled 'several times' (i.e., it remains enabled after initiating a firing), it may start several independent firings in the same time instant.

The firing times of some transitions may be equal to zero, which means that the firings are instantaneous; all such transitions are called immediate (while the other are called timed). Since the immediate transitions have no tangible effects on the (timed) behavior of the model, it is convenient to 'split' the set of transitions into two parts, the set of immediate and the set of timed transitions, and to fire first the (enabled) immediate transitions; only when no more immediate transitions are enabled, the firings of (enabled) timed transitions are initiated (still in the same instant of time). It should be noted that such a convention effectively introduces the priority of immediate transitions over the timed ones, so the conflicts of immediate and timed transitions should be avoided. Also, the free–choice classes of transitions must be 'uniform', i.e., all transitions in each free–choice class must be either immediate or timed.

The firing times of transitions can be either deterministic or stochastic (i.e., described by a probability distribution function); in the first case, the corresponding timed nets are referred to as D–timed, in the second, for the (negative) exponential distribution of firing times, the nets are referred to as M–timed (or Markovian) nets. In both cases, the concepts of states and state transitions have been formally defined and used in derivation of different performance characteristics of the model [Zu91].

Timed Petri net models are discrete–event systems as the changes of model 'states' (as the result of transition firings) occur in discrete instants of time (for both instantaneous and timed firings). Analysis of timed models by using discrete–event simulation is thus a 'natural' approach to model evaluation, which imposes very few restrictions on the class of

analyzed models (other approaches, like reachability analysis and structural analysis, can be applied only to certain classes of net models).

In discrete–event simulation, an occurrence of an event may cause some other events to happen at the same time or at some future time. There are two basic methods of organizing discrete–event simulations [Mo89], the time–based (or synchronous–timing [Ko81] or fixed–step [Fe78]) simulation and event–driven (or asynchronous–timing [Ko81]) simulation. For time–based simulation, the model is analyzed at consecutive, uniformly distributed time instants (the time–step is constant), and all events which can occur at these time instants, are executed (changing the state of the model); although this approach is rather simple to implement, quite often it also is very inefficient, especially when events are clustered in time (i.e., there are periods of high activities followed by periods of inactivities of the model).

In event–driven simulation, the control of the (simulated) time depends only upon the activities of the model. During execution of events, all future events are stored in a list of events also called the 'event queue'. This list is ordered with respect to the time in which the events are scheduled to occur; the event scheduled to occur in the nearest future is at the front of the list. If there are no more events to be executed at the present (simulated) time instant, the 'first' event is fetched (or 'dequeued') from the list of events, the (simulated) time is advanced accordingly, and the event is executed (possibly creating new events). Event–driven simulation is more difficult to implement than time–based simulation, but it is much more flexible with respect to time control, as it analyzes the model only at time instants when events occur.

Let `schedule(Event,Time)` denote the operation of "inserting", in the event queue, a new event of type `Event` which is scheduled to occur at time instant `Time`. A general outline of the event–driven simulation can be as follows:

```
schedule(InitialEvent,0.0);
while nonempty(EventQueue) do
   dequeue(EventQueue,Event);
   SimulatedTime := Event.Time;
   execute(Event)
endwhile;
```

The simulation is usually executed for some fixed period of (simulated) time; let this period of time be denoted `TimeLimit`. This time limit can be incorporated in the simulator in one of two basic ways. One approach is to schedule a special event, `EndSimulation`, at time `TimeLimit`; execution of this event terminates the simulation and outputs simulation results:

```
schedule(EndSimulation,TimeLimit);
schedule(InitialEvent,0.0);
while nonempty(EventQueue) do
   dequeue(EventQueue,Event);
   SimulatedTime := Event.Time;
   execute(Event)
endwhile;
```

The second approach uses a boolean variable `Continue` to terminate the simulation loop when the value of `SimulatedTime` becomes greater than `TimeLimit`:

```
Continue := true;
schedule(InitialEvent,0.0);
while nonempty(EventQueue) and Continue do
   dequeue(EventQueue,Event);
   SimulatedTime := Event.Time;
   if SimulatedTime > TimeLimit then Continue := false
   else execute(Event)
endwhile;
```

It should be observed that if no new events are created during execution of the initial event (or subsequent events), the simulation run ends rather quickly.

The results of simulation runs may include different performance measures, evaluated and presented in many different ways. Quite often these results may depend upon a particular application. Therefore only very simple data are collected during the simulation of net models, and some 'postprocessing' capabilities are provided to calculate performance measures from these 'raw data'.

SIMULATION

Simulation of the behavior of net models is performed by the directive:

```
simulate(tlimit);
```

where `tlimit` is the simulation time limit (in the same units as the firing times of transitions). During the simulation, the numbers of firings are counted for each occurrence of each transition, and also the total time of all firings (including overlapping firings) is cumulated for each occurrence. For D–timed firings, the firing times are constant, as specified in the net description; for M–timed firings, the firing times are generated by a random number generator (with exponential distribution), using the average firing time as a (scaling) parameter. Moreover, the numbers of tokens entering each place are also counted (including the initial marking), and the total 'waiting' time of all tokens is cumulated for each place (and color). After the termination of simulation, these results of the simulation can be obtained by the directive:

```
simres;
```

which outputs a list of firing counts and total firing times for all 'active' occurrences (i.e., occurrences with nonzero firing counts); for immediate occurrences, only firing counts are provided. Similarly, the token counts are output for all 'active' places together with the total (nonzero) waiting times.

Example 1. A D–timed Petri net model of a simple protocol with a timeout mechanism [Zu96] is shown in Fig.1.

The token in p_1 represents a message which a 'sender' (p_1) sends to a 'receiver' (p_3) and which is confirmed by an acknowledgement sent back to the sender. The message is

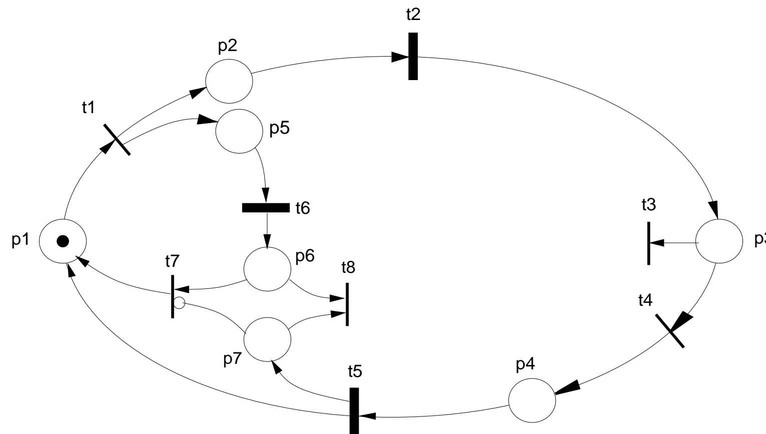

Fig.1. A simple protocol with a timeout.

sent by firing t_1, after which a single token is deposited in p_2 (the message) and in p_5 (the timeout). Firing time of t_2 represents the 'communication delay' of sending a message, and firing time of t_6, the timeout time. When the firing of t_2 is finished, a token is deposited in p_3, the receiver. p_3 is a free-choice place, so t_3 and t_4 are enabled simultaneaously, but only one of them can fire; the random choice is characterized by 'choice probabilities' assigned to t_3 and t_4. t_3 represents (in a simplified way) the loss or distortion of the message or its acknowledgement; it t_3 is selected for firing (according to its free–choice probability), the token is removed from p_3 as well as from the model (t_3 is a 'token sink'). In such a case the timeout transition t_6 will finish its firing with no token in p_7, t_7's firing will regenerate the lost token in p_1, and the message will be 'retransmitted'. If the message is received correctly, t_4 is selected for firing rather than t_3, and after another 'communication delay' (modeled by t_5), tokens are deposited in p_7 and p_1 (so another message can be sent to the receiver). The token in p_7 waits until the timeout transition t_6 finishes its firing, and then removes the timeout token by firing t_8 (t_7 is disabled in this case by the inhibitor arc).

Note: The transition t_4 may seem redundant in this model but in fact it is required due to the restriction that all free–choice classes of transitions must be uniform, i.e., each free–choice class must contain either immediate or timed transitions but not both.

All immediate transitions (i.e., transitions with zero firing time) are represented by (thin) bars, while timed transitions are represented by (black) rectangles. The firing times of timed transitions are selected in such a way that the timeout time (t_6) is greater than the sum of the delays of sending a message (t_2) and an acknowledgement (t_5). In the following description, the immediate transitions are indicated by the default zero firing time:

```
Dnet(#1=1/2,5;
     #2*5=2/3;
     #3,1/10=3;
     #4,9/10=3/4;
     #5*5=4/1,7;
     #6*15=5/6;
```

```
     #7=6,7:0/1;
     #8=6,7)
mark(1);
simulate(10000);
simres;

Simulation Counts:
 firings of #1 : 956
 firings of #2 : 956 total firing time 4780
 firings of #3 : 88
 firings of #4 : 868
 firings of #5 : 868 total firing time 4340
 firings of #6 : 956 total firing time 14340
 firings of #7 : 88
 firings of #8 : 867

 tokens in 1 : 956
 tokens in 2 : 956
 tokens in 3 : 956
 tokens in 4 : 868
 tokens in 5 : 956
 tokens in 6 : 955
 tokens in 7 : 867 total waiting time 4335
```

The results show that the numbers of firings of t_3 and t_7 are approximately ten times smaller than the number of t_4's firings, and that the average waiting time in p_7 is equal to 5.0, which is exactly the difference between the timeout time (15.0) and two communication delays (5.0+5.0).

In the model shown in Fig.1, the timeout is not really 'canceled'; it is 'blocked' by the inhibitor arc (p_7, t_7) (and then removed by t_8). A more "realistic" modeling of the timeout uses an interrupt arc [Zu96], as shown in Fig.2, with the following results:

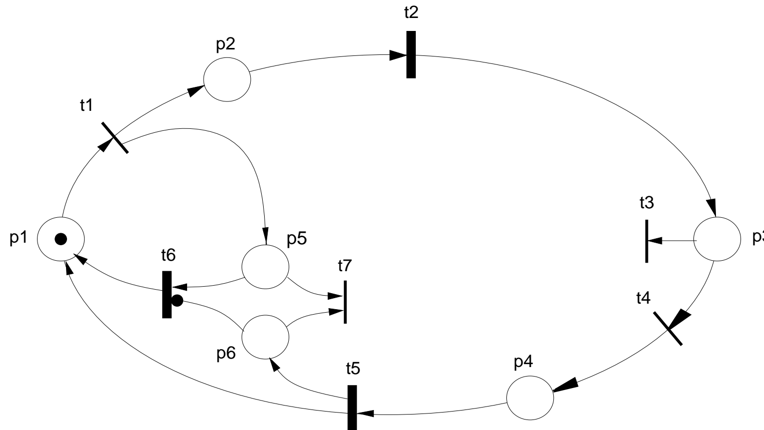

Fig.2. A timeout model with an interrupt arc.

```
Dnet(#1=1/2,5;
     #2*5=2/3;
     #3,1/10=3;
     #4,9/10=3/4;
     #5*5=4/1,6;
     #6*15=5,6-/1;
     #7=5,6)
mark(1);
simulate(10000);
simres;

Simulation Counts:
 firings of #1 : 956
 firings of #2 : 956 total firing time 4780
 firings of #3 : 88
 firings of #4 : 868
 firings of #5 : 868 total firing time 4340
 firings of #6 : 956 total firing time 10005
 firings of #7 : 867

 tokens in 1 : 956
 tokens in 2 : 956
 tokens in 3 : 956
 tokens in 4 : 868
 tokens in 5 : 1823
 tokens in 6 : 867
```

The increased number of tokens in p_5 is due to interrupted firings of t_6 (the tokens re–entering the places are also counted); these interrupted firings also result in reduced total firing time of t_6.

Example 2. A colored Petri net model of ‘five dining philosophers’ [Zu96] is shown in Fig.3. The philosophers are represented by token colors A, B, C, D and E, while forks by colors m, n, o, p and q.

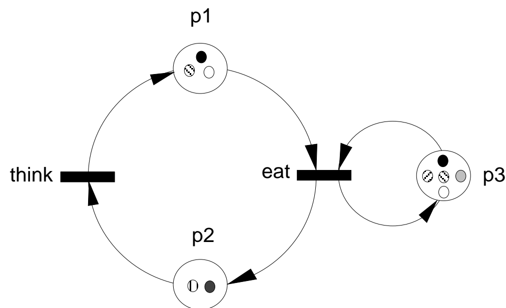

Fig.3. A colored net model of ‘dining philosophers’.

The net shown in Fig.3 outlines a model of a single philosopher, so the use of colors is essential for this model. The use of colored tokens and the conflicts created by sharing forks can be described by the following ‘connectivity matrix’, in which rows correspond to

colors of tokens assigned to places, and columns – to transition occurrences; the elements "+1" represent arcs from occurrences to places while elements "-1", arcs form places to occurrences (the values of the elements are the weights assigned to arcs, in this case they are equal to 1). The elements of the matrix are sets rather than expressions, so the 'loops' (e.g., place p_3 and transition *eat* in Fig.3) can also be represented:

	think					*eat*				
	U	V	W	X	Y	U	V	W	X	Y
p_1:A	+1					-1				
B		+1					-1			
C			+1					-1		
D				+1					-1	
E					+1					-1
p_2:A	-1					+1				
B		-1					+1			
C			-1					+1		
D				-1					+1	
E					-1					+1
p_3:m						-1,+1	-1,+1			
n							-1,+1	-1,+1		
o								-1,+1	-1,+1	
p									-1,+1	-1,+1
q						-1,+1				-1,+1

For example, the last column describes the occurrence `Y` of transition *eat*. The occurrence is enabled (element "-1") by a single token of color E in place p_1 (philosopher E), one token of color p and one of color q in place p_3 (E's 'right' and 'left' forks). Firing this occurrence removes these tokens from p_1 and p_3 and when the eating is finished (the firing time of this occurrence), one token of color E is deposited in p_2 (philosopher E is goint to think), and single tokens of color p as well as q are returned to p_3 (the two forks are returned). It should be observed that the name of the occurrence, `Y`, is rather irrelevant; since this occurrence models an 'action' of philosopher E, it could be named `E` as well.

The description of the model and simulation results can be as follows (all firing times are exponentially distributed in this example):

```
color(A,B,C,D,E,m,n,o,p,q);
Mnet(#think*5{U=2:1A/1:1A},
     {V=2:1B/1:1B},
     {W=2:1C/1:1C},
     {X=2:1D/1:1D},
     {Y=2:1E/1:1E};
#eat*2{U,1=1:1A,3:1q,3:1m/2:1A,3:1q,3:1m},
     {V,1=1:1B,3:1n,3:1m/2:1B,3:1n,3:1m},
     {W,1=1:1C,3:1n,3:1o/2:1C,3:1n,3:1o},
     {X,1=1:1D,3:1o,3:1p/2:1D,3:1o,3:1p},
     {Y,1=1:1E,3:1q,3:1p/2:1E,3:1q,3:1p})
mark(1:1A,1:1B,1:1C,2:1D,2:1E,3:1m,3:1n,3:1o,3:1p,3:1q);
```

```
simulate(10000);
simres;

Simulation Counts:
 firings of #eat.U : 1182 total firing time 2436.41
 firings of #eat.V : 1214 total firing time 2452.88
 firings of #eat.W : 1230 total firing time 2404.69
 firings of #eat.X : 1264 total firing time 2374.24
 firings of #eat.Y : 1242 total firing time 2467.94
 firings of #think.U : 1182 total firing time 6041.32
 firings of #think.V : 1214 total firing time 6022.06
 firings of #think.W : 1230 total firing time 6148.03
 firings of #think.X : 1265 total firing time 6275.35
 firings of #think.Y : 1243 total firing time 6102.78

 tokens in 1:A : 1182 total waiting time 1468.68
 tokens in 1:B : 1214 total waiting time 1451.64
 tokens in 1:C : 1230 total waiting time 1411.45
 tokens in 1:D : 1264 total waiting time 1331.9
 tokens in 1:E : 1242 total waiting time 1349.8
 tokens in 2:A : 1182
 tokens in 2:B : 1214
 tokens in 2:C : 1230
 tokens in 2:D : 1265
 tokens in 2:E : 1243
 tokens in 3:m : 2397 remaining 1, total waiting time 5013.91
 tokens in 3:n : 2445 remaining 1, total waiting time 5038.67
 tokens in 3:o : 2495 remaining 1, total waiting time 5137.51
 tokens in 3:p : 2507 remaining 1, total waiting time 5070.98
 tokens in 3:q : 2425 remaining 1, total waiting time 4989.4
```

It should be observed that the average firing time of all occurrences of transition *eat* is approximately 2.0, and that of occurrences of transition *think* is approximately equal 5.0. Moreover, the total numbers of firings for all occurrences are approximately the same, so the conflict resolution is fair and unbiased. The numbers of tokens deposited in p_3 (for each color) are approximately twice as large as the numbers of tokens deposited in p_1 and p_2 (each fork is used by two philosophers). There are no waiting times for tokens in p_2.

Example 3. The net shown in Fig.4 is a simple illustration of 'marking–dependent' conflict resolutions [Zu96]. The net represents an interactive system executing two classes of jobs, say class–A and class–B jobs, with random selection of jobs from a common pool of waiting jobs. As no priotities and no queueing is assumed, the probability of selection of a class–A job (if there is any such job waiting) is determined by the ratio of the number of waiting class–A jobs to the total number of waiting jobs.

In the model, p_1 represents the (waiting) processor. Execution of class–A jobs is represented by t_2, and class-B jobs – by t_3. t_1 models the 'thinking time' for class–A jobs, and t_4 – the same for class–B jobs. p_2 is the pool of waiting class–a jobs, p_4 – the pool of waiting class–B jobs. t_2 and t_3 are in conflict because of sharing p_1, and the relative frequencies of t_2 and t_3 firings can be determined by the token counts in p_2 and p_4, respectively.

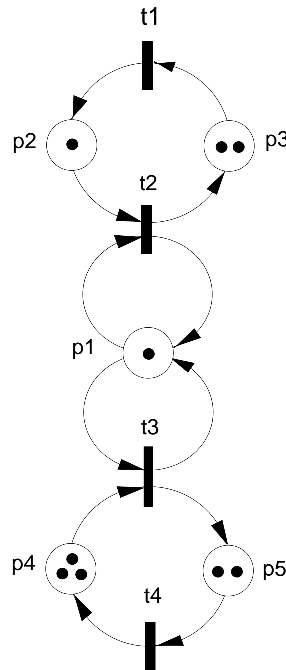

Fig.4. A processor executing two classes of jobs.

Assuming that all firing times are exponentially distributed, the description of the model and its simulation results are as follows:

```
Mnet(#1*5=3/2;
     #2*3,[2]=1,2/1,3;
     #3*2,[4]=1,4/1,5;
     #4*8=5/4)
mark(1,2,3:2,4:3,5:2);
simulate(10000);
simres;

Simulation Counts:
 firings of #1 : 1707 total firing time 8517.73
 firings of #2 : 1705 total firing time 5021.65
 firings of #3 : 2392 total firing time 4793.9
 firings of #4 : 2393 total firing time 18836.8

 tokens in 1 : 4097 total waiting time 44.311
 tokens in 2 : 1705 total waiting time 16411.8
 tokens in 3 : 1707
 tokens in 4 : 2393 remaining 1, total waiting time 26298.5
 tokens in 5 : 2393
```

To check the effect of marking–dependent arcs, these simulation results can be compared with the case when the relative frequencies of (conflicting) firings are constant and are the same for both transitions (which is the default case):

```
 Mnet(#1*5=3/2;
      #2*3=1,2/1,3;
      #3*2=1,4/1,5;
      #4*8=5/4)
 mark(1,2,3:2,4:3,5:2);
 simulate(10000);
 simres;

Simulation Counts:
 firings of #1 : 1819 total firing time 9436.65
 firings of #2 : 1818 total firing time 5487.67
 firings of #3 : 2197 total firing time 4335.24
 firings of #4 : 2199 total firing time 17034.1

 tokens in 1 : 4015 total waiting time 37.7378
 tokens in 2 : 1819 remaining 1, total waiting time 15041.7
 tokens in 3 : 1819
 tokens in 4 : 2202 remaining 5, total waiting time 28512
 tokens in 5 : 2199
```

Intuitively, class–B jobs should be disadvantaged in this case because it might be expected that, on average, there will be more waiting class–B jobs (in p_4) than class–A jobs (in p_2), so with the same probabilities of selection, class–B jobs will be selected less often than in the first case (i.e., marking–dependent frequencies). The results confirm this observation, but a better verification method, with a quantitative prediction of results, is needed.

If the model is slightly modified by making the transitions t_1 and t_4 immediate (i.e., with zero firing times), t_2 will always be enabled by all three class–A tokens in p_2, and t_3 will always be enabled by all five class–B tokens in p_4. For a marking–dependent frequency of t_2 and default frequency of t_3 (i.e., 1), the total number of firings of t_2 should be approximately three times greater than that of t_3:

```
 Mnet(#1=3/2;
      #2*3,[2]=1,2/1,3;
      #3*2=1,4/1,5;
      #4=5/4)
 mark(1,2,3:2,4:3,5:2);
 simulate(10000);
 simres;

Simulation Counts:
 firings of #1 : 2644
 firings of #2 : 2643 total firing time 8110.03
 firings of #3 : 920 total firing time 1891.05
 firings of #4 : 922

 tokens in 1 : 3563
 tokens in 2 : 2645 remaining 2, total waiting time 21890.5
 tokens in 3 : 2644
 tokens in 4 : 925 remaining 5, total waiting time 48096.6
 tokens in 5 : 922
```

POSTPROCESSING

The ('raw') simulation results are stored in the internal structure representing the net [Zu96], so after a simulation run, the results of simulation can be used for evaluation of different performance measures. A 'postprocessing' interpreter is provided which evaluates simple expressions composed of constants, net parameters and simulation results. The interpreter is invoked by the command `comp` with an 'expression' as its argument:

```
<command> ::= comp ( <expr> ) ;
<expr> ::= <term> | <expr> + <term | <expr> - <term>
<term> ::= <fact> | <term> * <fact | <term> / <fact>
<fact> ::= <number> | <t-arg> | <p-arg> | <s-name> | ( <expr> )
<s-name> ::= STime | SimTime
<t-arg> ::= t ( <t-ident> ) | n ( <t-ident> ) |
            f ( <t-ident> ) | p ( <t-ident> )
<p-arg> ::= t ( <p-ident> ) | n ( <p-ident> ) |
            m ( <p-ident> )
<t-ident> ::= <t-id> | <t-id> . <o-id>
<t-id> ::= # <number> | # <name>
<o-id> ::= <number> | <name>
<p-ident> ::= <p-id> | <p-id> : <color>
<p-id> ::= <number> | <name>
<color> ::= <name>
```

The 'special name' (`s-name`) `STime` (or `SimTime`) denotes the simulation time (`tlimit` of the last `simulate` directive).

The `t(...)` arguments refer to the total (cumulative) firing times (of transitions/occurrences) or total token waiting times (in places). The `n(...)` arguments refer to the numbers of firings (of transitions/occurrences) or the numbers of tokens entering the places (including the initial marking). The `f(...)` arguments refer to the firing times of transitions/occurrences, and the `p(...)` arguments, to the choice probabilities or relative frequencies assigned to transitions/occurrences.

Example. Some simple values calculated from the previous simulation results (Example 2) are as follows:

```
   Mnet(#1*5=3/2;
        #2*3,[2]=1,2/1,3;
        #3*2,[4]=1,4/1,5;
        #4*8=5/4)
   mark(1,2,3:2,4:3,5:2);
   simulate(10000);

comp(t(1:A)/n(1:A));            (* average waiting time of A *)
   value : 1.24253
comp(t(1:B)/n(1:B));            (* average waiting time of B *)
   value : 1.19575
comp(t(1:C)/n(1:C));            (* average waiting time of C *)
   value : 1.14752
```

```
comp(t(1:D)/n(1:D));            (* average waiting time of D *)
   value : 1.05371
comp(t(1:E)/n(1:E));            (* average waiting time of E *)
   value : 1.08680
comp(1.0-(t(3:m)+t(3:n)+t(3:o)+t(3:p)+t(3:q))/(5*SimTime)); (* fork "utilization" *)
   value : 0.49499
```

Because the formulas can become quite complicated, a mechanism for declaring variables and evaluating (sub)expressions is also provided:

```
<command> ::= <declaration> | <assignment> | comp ( <expr> ) ;
<declaration> ::= var <list-of-variables> ;
<list-of-variables> ::= <name> | <list-of-variables> , <name>
<assignment> ::= <variable> := <expr> ;
<variable> ::= <name>
```

Variables (after assigning values to them) can be used in expressions in the same way as the 'special names'. All variables must be declared before the first use, and each declaration of variables erases all previous declarations.

IMPLEMENTATION ISSUES

The 'event queue' is implemented as a two–level list structure, with the first level representing the (future) time instants (in ascending order), and the second level, for each time instant, represents all firings which are scheduled to terminate at this time instant. The event descriptor for the event queue is:

```
struct event_list {
     event_list * next;
     double       time;       /* event time */
     f_list     * olist;      /* list of occurrences */
};
```

and the list of occurrences is composed of the descriptors:

```
struct f_list {
     f_list    * next;
     o_list    * onode;      /* occurrence */
     int         val;        /* degree of firing */
};
```

where `onode` is a link to a (firing) occurrence, and `val` stores the degree of firing, i.e., the number of firings initiated at the same time instant.

The organization of the event queue uses its own (dynamic) memory manegement in the sense that it maintains a list of 'released' descriptors (in fact, two lists, for `event_list` and `f_list` descriptors), and when a new descriptor is needed, a check is first made if a released

descriptor is available; only when all descriptors are used, a new descriptor is created from additionally allocated memory. This solution can be characterized by the following statistics showing the maximum length of the event queue (i.e., the total number of allocated event descriptors) and the total number of descriptors requests (i.e., the total number of time instants analyzed during the simulation):

Example	*max length*	*number of requests*
1	3	1913
2	6	7131
3	8	5557

For each time instant, three consecutive steps are performed; first, all firings which are scheduled for termination as the current event, are processed and the marking function is updated by depositing tokens to occurrences' output places (the simulation statistics for places are also updated at the same time).

In the second step, all possible firings of enabled immediate occurrences are executed (and the marking function is updated accordingly):

```
while nonempty(set_of_enabled_immediate_occurrences) do
   fire_all_enabled_conflict_free_occurrences;
   for each enabled_free_choice_class do
      randomly_select_an_occurrence;
      fire_the_selected_occurrence;
      ignore_remaining_occurrences_in_the_free_choice_class
   endfor;
   while nonempty(set_of_enabled_immediate_occurrences) do
      find_a_conflict_class_of_enabled_occurrences;
      randomly_select_an_occurrence;
      fire_the_selected_occurrence;
      ignore_remaining_occurrences_in_the_class
   endwhile
endwhile;
```

The third step initializes the firings of enabled timed occurrences (again updating the marking fuction):

```
while nonempty(set_of_enabled_timed_occurrences) do
   for each enabled_conflict_free_occurrence do
      initiate_occurrence_firing;
      determine_the_length_of_the_firing_time(FTime);
      schedule(occurrence,SimulatedTime+FTime)
   endfor;
   for each enabled_free_choice_class do
      randomly_select_an_occurrence;
      initiate_occurrence_firing;
      determine_the_length_of_the_firing_time(FTime);
      schedule(occurrence,SimulatedTime+FTime);
      ignore_remaing_occurrences_in_the_free_choice_class
   endfor;
```

```
    while nonempty(set_of_enabled_immediate_occurrences) do
       find_a_conflict_class_of_enabled_occurrences;
       randomly_select_an_occurrence;
       initiate_occurrence_firing;
       determine_the_length_of_the_firing_time(FTime);
       schedule(occurrence,SimulatedTime+FTime);
       ignore_remaining_occurrences_in_the_class
    endwhile
 endwhile;
```

The length of the firing time is determined depending upon the type of transition, occurrence or net. For D–timed firings, `FTime` is simply the firing time of the occurrnce; for M–timed firings, an exponentially distributed random number generator is used with the occurrence's average firing time as a parameter.

Identification of (enabled) conflict–free occurrences and (enabled) free–choice classes of occurrences is done during the initial processing of the net; all conflict–free occurrences have the `class` field in the descriptor [Zu96] set to `1`, the same field for free–choice classes of occurrences is set a value greater than `1` (the same value for all occurrences in the same free–choice class), and for all other occurrences (potentially in conflicts) `class` is set to `0`.

For free–choice classes of occurrences (assumed to be linked into a list `Class`), and for a conflict class (also linked into a list `Class`), the selection of an occurrence for firing (`randomly_select_an_occurrence`) uses a uniformly distributed random number generator `UniformRandomNumber`:

```
 prob := UniformRandomNumber;
 while nonempty(Class) do
    if prob < Class.prob then select(Class);
    prob := prob - Class.prob;
    next(Class)
 endwhile;
 error("unsuccessful selection.-");
```

Since the classes of conflicting occurrences are marking–dependent, they are determined for each marking function; assuming that all enabled occurrences are linked in a list `List`, and that the conflict class will be represented by another list, `Class`, initially the first occurrence from `List` is moved to `Class`, and then, iteratively, those occurrences from `List` which share any (colored) places with occurrences in `Class` are moved from `List` to `Class`:

```
 Class := head(List);
 List := tail(List);
 Continue := true;
 while Continue do
    Continue := false;
    for each OccurL in List do
       for each OccurC in Class do
          if nonempty(shared(OccurL.input,OccurC.input)) then
             delete(OccurL,List);
             insert(OccurL,Class);
```

```
          Continue := true
        endif
      endfor
    endfor
  endwhile;
```

where `shared(list1,list2)` determines the set of shared elements on the the lists `list1` and `list2`.

The process of identifying conflict classes for a given marking function, and in fact, for a given list of enabled occurrences, is rather time–consuming. Therefore, instead of repeatedly checking possible conflicts by comparing the input lists of occurrences, it may be beneficial to save the performed decompositions of the original lists `List` into `Class` and the remaining list `List` (which may be empty), and to reuse these saved decomposition if the same (original) list `List` needs to be analyzed again.

The simulation module can save and reuse the decompositions. This option can be activated by a directive:

```
conf;
```

and can be deactivated (i.e., turned off) by directive:

```
noconf;
```

The decompositions are saved in a (dynamically created) tree–like structure in which consecutive layers of the tree are selected by consecutive occurrences of the (ordered) original list of (enabled) occurrences `List`, and in which the nodes are associated with pairs of decomposed list `Class` and new `List` (which can be further decomposed); the descriptor of each node in this structure is:

```
struct dcomp_list {
  dcomp_list * next;     /* next element */
  dcomp_list * down;     /* next level */
  o_list * onode;        /* occurrence */
  f_list * conf;         /* conflict class */
  f_list * tail;         /* remaining list */
};
```

There is a limit on the (total) number of nodes in the decomposition tree. If this limit is exceeded, the `conf` option is automatically turned off and the (partial) decomposition tree is deleted. The default value of this limit is `1000` nodes, and it can be redefined to a new value `val` by the directive:

```
dlimit=val;
```

Moreover, the created decomposition tree can be displayed using the directive:

```
decomp;
```

Note: It should be noted that the `conf` option is justified only when the set of possible marking functions is rather small, so there is a good chance of reusing the decompositions. If this is not the case, the use of this option can actually extend the simulation time as many new decompositions need to be stored, consuming large amounts of memory and performing many useless searches of the decomposition tree.

Example: The use of the `conf` option is shown for the model in Fig.3. The listing of the decomposition tree shows the levels of the structure and the actual decompositions enclosed in brackets '[' and ']':

```
color(A,B,C,D,E,m,n,o,p,q);
Mnet(#think*5{U=2:1A/1:1A},
     {V=2:1B/1:1B},
     {W=2:1C/1:1C},
     {X=2:1D/1:1D},
     {Y=2:1E/1:1E};
#eat*2{U,1=1:1A,3:1q,3:1m/2:1A,3:1q,3:1m},
     {V,1=1:1B,3:1n,3:1m/2:1B,3:1n,3:1m},
     {W,1=1:1C,3:1n,3:1o/2:1C,3:1n,3:1o},
     {X,1=1:1D,3:1o,3:1p/2:1D,3:1o,3:1p},
     {Y,1=1:1E,3:1q,3:1p/2:1E,3:1q,3:1p})
mark(1:1A,1:1B,1:1C,2:1D,2:1E,3:1m,3:1n,3:1o,3:1p,3:1q);
conf;
simulate(10000);
decomp;

#eat.U(#eat.V[eat.U,eat.V + NIL]
             (#eat.W[eat.U,eat.V,eat.W + NIL]
                    (#eat.X[eat.U,eat.V,eat.W,eat.X + NIL]
                           (#eat.Y[eat.U,eat.V,eat.W,eat.X,eat.Y + NIL]),
                     #eat.Y[eat.U,eat.V,eat.W,eat.Y + NIL]),
              #eat.X[eat.U,eat.V + eat.X]
                    (#eat.Y[eat.U,eat.V,eat.Y,eat.X + NIL]),
              #eat.Y[eat.U,eat.V,eat.Y + NIL]),
       #eat.W[eat.U + eat.W]
             (#eat.X[eat.U + eat.W,eat.X]
                    (#eat.Y[eat.U,eat.Y,eat.X,eat.W + NIL]),
              #eat.Y[eat.U,eat.Y + eat.W]),
       #eat.X[eat.U + eat.X]
             (#eat.Y[eat.U,eat.Y,eat.X + NIL]),
       #eat.Y[eat.U,eat.Y + NIL]),
#eat.V(#eat.W[eat.V,eat.W + NIL]
             (#eat.X[eat.V,eat.W,eat.X + NIL]
                    (#eat.Y[eat.V,eat.W,eat.X,eat.Y + NIL]),
              #eat.Y[eat.V,eat.W + eat.Y]),
       #eat.X[eat.V + eat.X]
             (#eat.Y[eat.V + eat.X,eat.Y]),
       #eat.Y[eat.V + eat.Y]),
#eat.W(#eat.X[eat.W,eat.X + NIL]
             (#eat.Y[eat.W,eat.X,eat.Y + NIL]),
```

```
      #eat.Y[eat.W + eat.Y]),
#eat.X(#eat.Y[eat.X,eat.Y + NIL])
```

The decomposition tree is also sketched in Fig.5 where nodes without any decomposition are shown by 'white' circles, and the nodes associated with decompositions – by dark circles. The list to be decomposed is represented by the path from the root; for example, the list (*eat*.U, *eat*.W, *eat*.X) is decomposed into (see the listing above) a single–element class (*eat*.U) and the remaining list (*eat*.W, *eat*.X); this remaining list (analyzed in another – possibly subsequent – decomposition) is a two–element conflict class (so it cannot be further decomposed).

There are 25 decompositions and 30 nodes in this decomposition tree. In the simulation shown above, the occurrences of transition *eat* are fired several thousand times, so without using the `conf` option, the same conflicts must be (unnecessarily) analyzed thousands of times.

Fig.5. The decomposition tree for the net shown in Fig.3.

CONCLUDING REMARKS

An event–driven simulation tool, recently added to *TPN–tools*, provides a simple and quite flexible means for charactering the behavior of timed models. The tool can be especially useful for evaluations of models for which other methods (for example, reachability analysis) cannot be used because of the size of the state space or unsatisfied constraints (for example, unboundedness of models).

At present, there are only two types of temporal information that can be associated with transitions and their occurrences: constant firing times and exponentially distributed firing times. Several other distributions can easily be implemented by minor entensions to the model description language [Zu96]. For example, additional (to `D`, `M` and `X`) transition and occurrence *types* can be defined, possibly including a 'user–defined' type (with some sort of standardized interface), and so on. Similarly, more elaborate forms of simulation results can be adopted from other simulation languages.

The described simulation tool was used extensively in several evaluation studies, including a simulation of a multilayered model of a communication network [Re96]. Its performance and flexibility compared quite favorably with some other Petri net tools used for similar studies (for example, it was an order of magnitude faster and much more flexible than Visual SimNet [Re96]).

References

[Aa93] van der Aalst, W.M.P., "Interval timed colored Petri nets and their analysis"; in: "Applications and Theory of Petri Nets 1993" (Lecture Notes in Computer Science 691); Ajmone Marsan, M. (ed.), pp.453–472, Springer Verlag 1993.

[Ag79] Agerwala, T., "Putting Petri nets to work"; IEEE Computer Magazine, vol.12, no.12, pp.85-94, 1979.

[AM84] Ajmone Marsan, M., Conte, G., Balbo, G., "A class of generalized stochastic Petri nets for the performance evaluation of multiprocessor systems"; ACM Trans. on Computer Systems, vol.2, no.2, pp.93–122, 1984.

[Fe78] Ferrari, D., "Computer systems performance evaluation"; Prentice–Hall 1978.

[Ko81] Kobayashi, H., "Modeling and analysis – an introduction to system performance evaluation methodology"; Addison–Wesley 1981.

[MF76] Merlin, P.M., Farber, D.J., "Recoverability of communication protocols – implications of a theoretical study"; IEEE Trans. on Communications, vol.24, no.9, pp.1036–1049, 1976.

[Mo82] M.K. Molloy, "Performance analysis using stochastic Petri nets"; IEEE Trans. on Computers, vol.31, no.9, pp.913–917, 1982.

[Mo89] Molloy, M.K., "Fundamentals of performance modelling"; Macmillan 1989.

[Mu89] Murata, T., "Petri nets: properties, analysis and applications"; Proceedings of IEEE, vol.77, no.4, pp.541–580, 1989.

[Pe81] Peterson, J.L., "Petri net theory and the modeling of systems"; Prentice–Hall 1981.

[Re96] Reid, M., "Modeling and performance analysis of ATM LANs"; M.Sc. Thesis, Department of Computer Science, Memorial University of Newfoundland, St. John's, Canada A1B 3X5, 1996.

[Re85] Reisig, W., "Petri nets - an introduction" (EATCS Monographs on Theoretical Computer Science 4); Springer Verlag 1985.

[Zu91] Zuberek, W.M., "Timed Petri nets – definitions, properties and applications"; Microelectronics and Reliability (Special Issue on Petri Nets and Related Graph Models), vol.31, no.4, pp.627–644, 1991 (available through anonymous `ftp` at `ftp.cs.mun.ca` as `/pub/publications/91-MaR.ps.Z`).

[Zu96] Zuberek, W.M., "Modeling using timed Petri nets – model description and representation"; Technical Report #9601, Department of Computer Science, Memorial University of Newfoundland, St. John's, Canada A1B 3X5, 1996 (available through anonymous `ftp` at `ftp.cs.mun.ca` as `/pub/techreports/tr-9601.ps.Z`).