International Journal
Information Technology Applications (ITA)
Volume 9, Number 1, September 2020

DEMONSTRATIVE MODELING OF DISCRETE DYNAMIC SYSTEMS

Juraj Štefanovič

Abstract:

Modeling and simulation of discrete dynamic systems includes several basic principles. Here the principles of event-driven graphs, queueing systems and Petri-nets were chosen. Three simple implemented models for study and experimental purposes were created. These models are implemented as a short source code in C-language, including the simulation engine without any black-box parts for simple overview of everything from abstract principle to the bottom code. Simulation models can be extended but these simple examples can already produce complex behavior.

Keywords:

Event driven graphs, queueing systems, Petri nets, discrete dynamic systems, practices in C-language.

ACM Computing Classification System:

Modeling and simulation.

Introduction

Discrete dynamic systems are subject of study for many diverse purposes in technology of control systems and production management. Our intention was to create not too complicated models of discrete dynamic systems for study and education. First model is based on the principle of event-driven graphs, they were introduced by classical work [1] and explained in various sources, as for example [2], [4]. In our region is to mention the interactive graphic tool [3] which was based on the open-source C++ library "Tiny". This library was created for study purposes, with relative rich content including event-driven structures, queues, simulation control modules, random number generators, etc. Second model here is an extension of the first one, including the principle of queueing systems. Queueing system is an abstract model with many practical representations and explanations wherever [5]. The third presented model is a simple representation of production line as an example of control with external events, they are given from the user and/or from automatic control which is constructed by Petri-net. Control systems based on Petri-nets are well known and studied, for example [6]. Our result here is the creation of easy to understand implementations of simulation examples, they are suitable for study and experimental purposes.

1 Event-Driven Model with Basic Implementation

To start with event-driven model, there is some very basic example of dynamic system: people are coming into entrance hall and a lift operates to take them away. Two different events occur in parallel, when model is running, see (Fig.1). **`Event1`** is that another person comes and increments the amount of waiting people. **`Event2`** is the operation of lift, which takes several people with and decrements the amount of people. Both events are scheduled itself repeatedly to run after some random number of timesteps. This model contains **`event0`** which runs only once at the very start of simulation to make first scheduling of events 1 and 2.

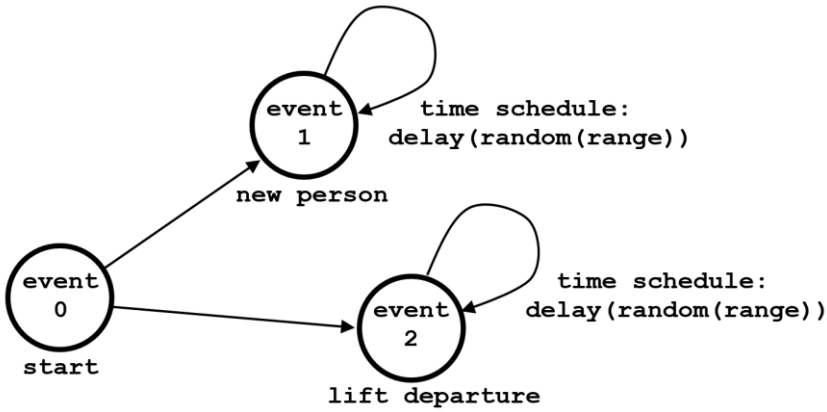


Fig.1. Event-driven system: persons are coming into entrance hall and lift operates.

An event-driven model as shown above can be implemented with the use of usual programming language, when the event-calendar is prepared. Event-calendar is a simple list of records {time, event}. Time is a number of simulation timestep when event (an identification number of event) is scheduled to run.

We use examples in C-language. Optimal implementation of the event calendar is some object in C++ with private data and public methods to enter and remove these data. For the purposes of basic C-language training, we can write each method as an individual function and the calendar data is some global structure of variables, static allocated in the easiest case:

```
void calendar_initialise(void);
// make empty queue of records in the calendar data structure

void calendar_print(void);
// dump contents of calendar records for debugging purposes

void event_schedule(int delay, int event)
// add the record {current_time + delay, event} to calendar

int event_cause(void)
// proceed the simulation_time to the next record {time, event}
// delete this record from queue and return the number event
```

The simulation loop is some kind of decision switch, what event is to execute at the next simulation step. This basic implementation can be extended deliberately with another events and more complicated behavior.

```
event_schedule(1,1);       // first events must be scheduled
event_schedule(1,2);
while(simtime < maxtime)   // simulation loop with some constraint
  {
  event = event_cause();   // next event?
  printf("\ntime: %2d  event: %d  ", simtime, event);
  if(event==0) break;      // empty calendar: stop
  if(event==1) event1();
  if(event==2) event2();
  }
```

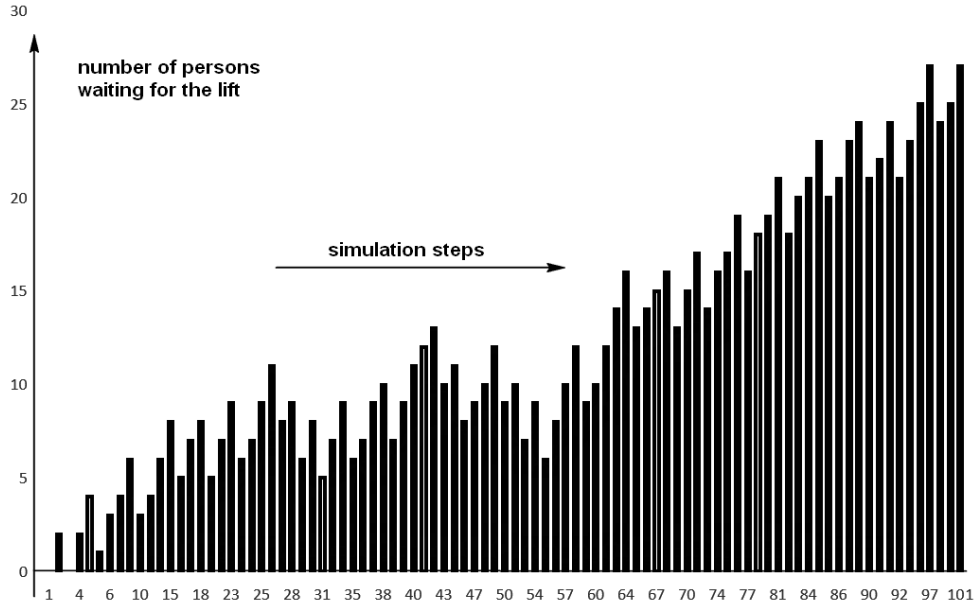
Having prepared the functionality of the event calendar, the model of entrance hall and a lift can be implemented in some first version as follows:

```
void event1(void)              // people are coming to the hall
  {
  int i;
  i = get_random_number(1,3);  // 1 - 3 persons come
  printf("persons came: %d", i);
  persons = persons + i;
  event_schedule(get_random_number(2,5),1);
  // this event 1 (next persons) returns after 2.- 5 timesteps
  return;
  }

void event2(void)              // lift departures from the hall
  {
  persons = persons - 3;
  if(persons < 0) persons = 0; // max 3 people enter the lift
  printf("lift is left, waiting persons: %d", persons);
  event_schedule(get_random_number(3,5),2);
  // this event 2 (lift) returns after 3 - 5 timesteps
  return;
  }
```

Even though this mentioned model is simple with two not complicated events, the behavior might be interesting as well. On (Fig.2) there is an example of simulation run, which shows some sensitivity to random deviations. The number of waiting people can be relatively stable in some range of time, but an overflow may blow without prevision. Changing parameters of model with study of its stable and unstable behavior is possible.

Fig.2. An example of simulation output.



2 Model Extension with Queueing System

Previous model contained one queue of waiting persons, but that queue was interpreted in easy way as a set, containing only a number of persons without their ordering. The next model is extended with the implementation of queues. In (Fig.3) some *transactions* (persons, products, administration letters, data packets, etc.) are coming by **event1** and placed in order into first waiting **queueA**. Each transaction is represented as an unique integer. **Event2** is the activity of first service place, where some decision is made to what queue the current transaction is placed next. In this model, several parameters must be set in advance: arrivals of transactions, time of each service and the decision policy between **queueB** and **queueX**.

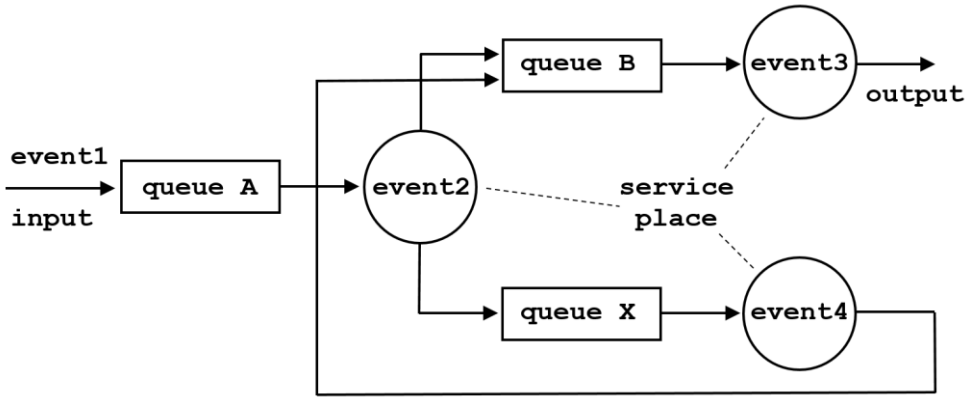


Fig.3. Queueing system with three queues and three service places.

Four events are implemented in similar way with scheduling statements as in previous model example. The **event2** makes some trivial decision where to put current transaction: if the number is even or odd.

```
void event1(void)          // transactions are coming to the queueA
  {
  static int item = 0; item++; // each transaction as unique number
  (void)queueA(item);          // insert number into queueA
  event_schedule(10,1);        // next event1 in 10 timesteps
  }

void event2(void)              // transactions are taken from queueA
  {
  int item;
  item = queueA(-1);           // take an item from queueA
  if(item >= 0)                // if the queue is not empty
    {
    if(item % 2 == 0)  (void)queueB(item);  // put into queueB
    else               (void)queueX(item);  // put into queueX
    }
  event_schedule(10,2);        // next event2 in 10 timesteps
  }
```

```
void event3(void)  // transactions are taken from queueB to output
  {
  int item;
  item = queueB(-1);        // take an item from queueB
  if(item >= 0)             // not empty queue
    printf("  %d from B to out", item);  // give some info
  event_schedule(10,3);     // next event3 in 10 timesteps
  }

void event4(void) {...      // from queueX to queueB in similar way
```

An easiest way to implement queue (when somebody starts with C-programming) is to write a function with static array inside (**`static int qu[]`**) and with input and output using the function parameters and return statement. Negative numbers as parameters represent control input, positive numbers and zero are transactions being added into the queue.

```
int queueA(int insert)
// insert: 0..N insert a transaction as a number,
// -1 remove transaction and return it, -2 print, -3 initialize
  {
  static int qu[LENGTH];   // queue as an array of numbers
  if(insert >= 0) ...      // add transaction to the queue
  if(insert == (-1)) ...   // remove transaction from queue
  if(insert == (-2)) ...   // print the contents of queue
  if(insert == (-3)) ...   // initialize (reset) the queue
  return(...);         // transaction number or other information
  }
```

Certainly, each queue must be implemented this way as a separate function. In C++, separate object instances will implement this task more effective. The simulation loop is similar as in the previous model:

```
while(simulation_time < maximal_time) // simulation loop
  {
  event = event_cause();  // get event and shift time
  if(event==0) break;     // empty calendar, stop
  if(event==1) event1();
  if(event==2) event2();
  if(event==3) event3();
  if(event==4) event4();
  }
```

Behavior of this model is more complicated with more parameters then the previous model in the first part of paper. Experimental change of parameters allows to observe and study various effects. An example of observation, when parameters were setup as follows:

next event 1, 2, 3 and 4 were scheduled after 5, 6, 14 and 30 timesteps
a decision in **`event2`**: **`if(get_random_number(1,100)> 50)`** put into **`queueB`**

Here is a state of queues **`A,X,B`** with waiting transactions in some simulation timestep. Some transactions are randomly selected to make delayed bypass around **`queueX`**.

```
queue A: 39,38,37,36,35,34 >>>
queue X: 33,32,31,28,23,22,21,19,18,16 >>>
queue B: 15,30,29,27,26,12,25,24,10 >>>
```

3 Production Line with Petri Net Control

The next model is event-driven with events generated from outside, by manual control and/or automatic control. Production line (Fig.4) consists from seven machines they can be switched on and off by commands. Two transporters and two operation places have predefined time duration to complete their action. Three feeders move objects from place to place and their action is completed instantly in one current timestep of simulation. Six sensors return binary information, whether an object is present/absent at given place.

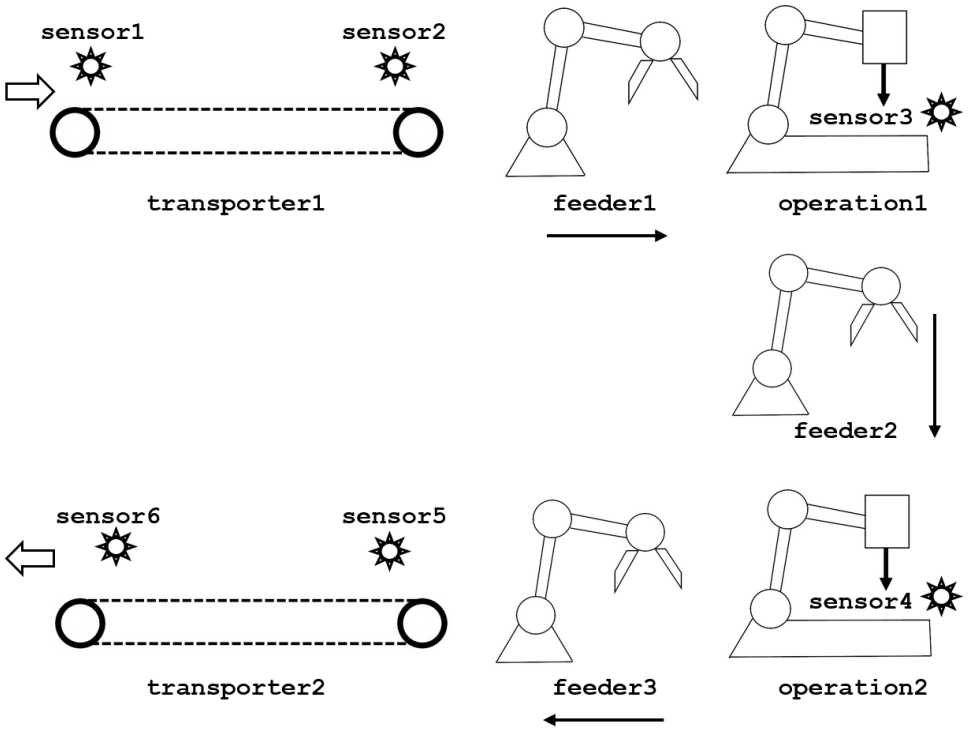


Fig.4. A production line model with 6 binary sensors and 7 machines under control.

This is the simulation loop:

```
link_reset();              // setup the data structure
while(1)
  {
  link_timestep();         // decrement all counters of work
  printf("\n\n\n  step %d: ", step++ );
  gets(line);              // read Enter key and command if any
  if(line[0]=='e') break; // the end, if command 'e'
  manual_control(line);    // intervention of person if command
  automatic_control();     // intervention of automatic control
  sensors_output();
  link_print();            // show state (Fig.5)
  }
```

On (Fig.5) there is a simple terminal-like visualisation of the state of production line. **Transporter1** is in state running, where **T1** denotes that an object (if the object was placed there) has finished his route on transporter. **Sensor2** confirms that there is an object at the end of transporter. **Transporter2** is running, **T4** means that an object (if the object was placed there) must move three timesteps of simulation, to reach the state **T1** being at the end of transporter. **Sensor5** and **sensor6** are placed at both edges of transporter, not any object is there. **Operation2** contains an object, **sensor4** gives confirmation and **T3** means that two timesteps of operation are needed to complete the work and reach the state **T1** = work completed, object present. When the **feeder3** takes this object away, **operation2** comes to the state **T0**.

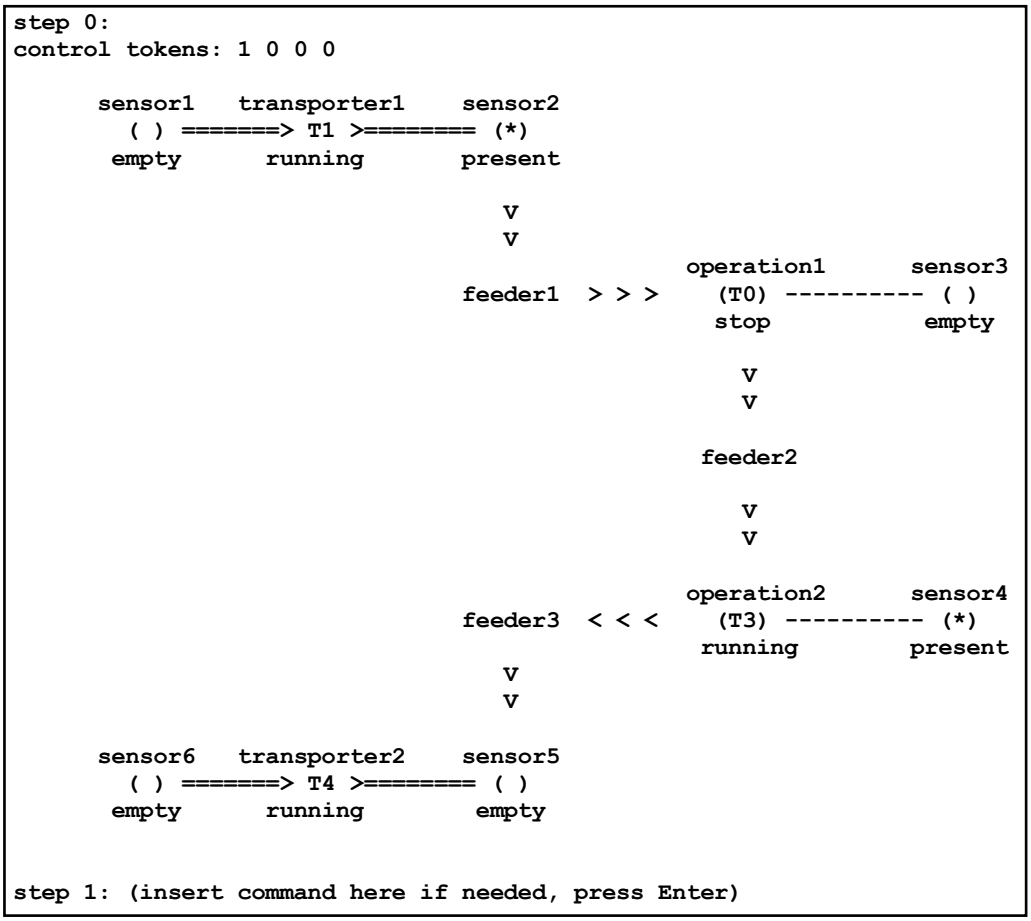


Fig.5. State of production line model as shown on text output.

Using this terminal-like interface, the production line can be controlled by these commands. At any timestep of simulation, only one command can be entered via terminal.

```
void manual_control(char *line)
  {
  if(line[0]=='i') insert_object();// to the first transporter
  if(line[0]=='t') take_object();  // from the second transporter
  if(line[0]=='p' && line[1]=='1') action_feeder1();
  if(line[0]=='p' && line[1]=='2') action_feeder2();
  if(line[0]=='p' && line[1]=='3') action_feeder3();
  if(line[0]=='d' && line[1]=='1') transporter1_on-off();
  if(line[0]=='d' && line[1]=='2') transporter2_on-off();
  if(line[0]=='o' && line[1]=='1') operation1_on-off();
  if(line[0]=='o' && line[1]=='2') operation2_on-off();
  }
```

A data structure contains the whole state information of the production line:

```
struct production_line
  {
  // sensors in state: object present/empty
  int sensor1; ...

  // transporters in state: stop/run
  int transporter1; ...

  // object on transporter:
  // 0 - absent, 1 - at the end, N - at the start,
  // N = decremented number of transport steps until 1 (finished)
  int transporter1_object; ...

  // operation in state: stop/run
  int operation1; ...

  // the operation place contains object:
  // 0 - absent, 1 - present, operation finished,
  // N = decremented number of operation steps until 1 (finished)
  int operation1_object; ...
  } link;
```

An example of action: feeder takes and inserts the object. Some error-detection is included into the code.

```
void action_feeder1(void)
// feeder completes his action at current simulation step
  {
  if(link.transporter1_object != 1)
    { printf("\n  error: object not found at the end"); return; }
  if(link.operation1_object > 0)
    { printf("\n  error: feeder1 disturbs operation1"); return(); }
  link.transporter1_object = 0; // take object from transporter
  link.operation1_object = N;   // insert into operation for N steps
}
```

An automatic control is constructed with the scheme of Petri net. On (Fig.6) there is a part of control scheme related to first transporter and feeder. When this example is used by practice lessons, students have to complete the control scheme to get some consistent solution. Using basic programming in C, the presence of tokens at all places of Petri net is saved into static array **`place[4]`**.

```
void automatic_control(void)
  {
  static int place[4] = { 1, 0, 0, 0 };
  // initial number of tokens in Petri net places

  if(place[0] == 1)         // if token at place1
    if(link.sensor1 == 1)     // if condition is true - object present
      {
      action_transporter1(); // transporter on
      place[0] = 0;           // get token from previous place
      place[1] = 1;           // put token to the next place
      }

  if(place[1] == 1)         // if token at place2
    if(link.sensor2 == 1)     // if condition is true - object present
      {
      action_transporter1(); // transporter off
      place[1] = 0;           // get token from previous place
      place[2] = 1;           // put token to the next place
      }

etc...
```

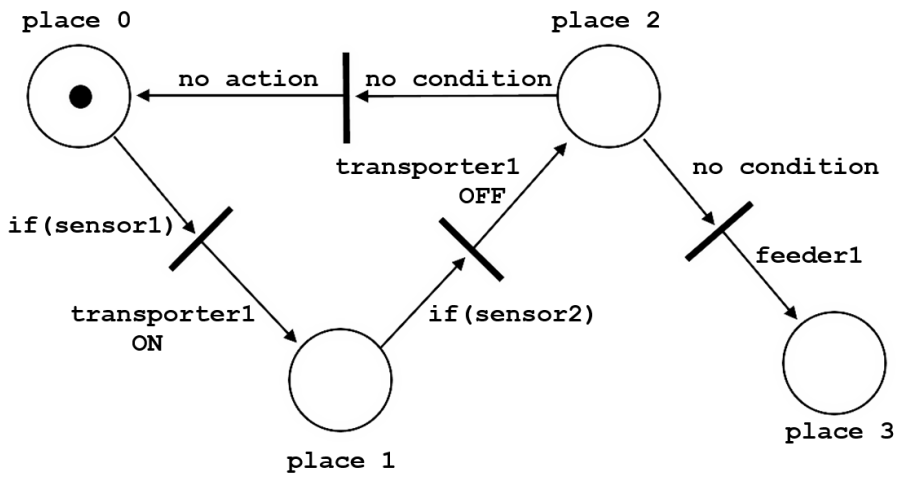


Fig.6. Petri net which controls first two machines from the model above.

Conclusion

Three examples of simple modeling with discrete event systems are presented. They were created for teaching and study purposes with following features:

- the whole model including the simulation engine is written in relative short source code using C-language, students can see the complete connection between an idea on the top (event-graphs, queue schemes, Petri-net) and the code on the bottom,
- there is not any black box inside, not only abstract training how to setup items on the screen and to press button to get simulation results,
- these models can be extended in any way, the code permits any change and extension
- despite of simplicity, small models are able to manifest complex or less predictable behavior, which is very instructive.

References

[1] Schruben, L.: Simulation Modeling with Event Graphs. *Communications of the ACM*, Volume 26, Number 11, November 1983, pp. 957-963. https://doi.org/10.1145/182.358460

[2] Leemis, L. M., Park, S. K.: *Discrete-Event Simulation: A First Course*. Upper Saddle River: Pearson Prentice Hall, 2006. 604 s. ISBN 0-13-142917-5.

[3] Šafařík, J., Netri, A., Solčány, V.: EGRET - Event Graph Editor and Simulator. 16th European Simulation Multiconference, ESM 2002, Darmstadt, Germany. In: *Proceedings of the 16th European Simulation Multiconference on Modeling and Simulation 2002*, ISBN 978-90-77039-07-6.

[4] MathWorks®, Help Center, *Create a Discrete-Event Model*. Online 8.2020. https://www.mathworks.com/help/simevents/gs/create-a-discrete-event-model.html

[5] *Queueing Systems, Theory and Applications*. Springer. ISSN 1572-9443/0257-0130. Online 8.2020 https://www.springer.com/journal/11134

[6] Giua, A., Seatzu, C.: Petri nets for the control of discrete event systems. Springer, *Software & Systems Modeling* 14, 693–701 (2015). https://doi.org/10.1007/s10270-014-0425-1

Authors

Ing. Juraj Štefanovič, PhD.
Faculty of Informatics, Pan-European University, Bratislava, Slovakia
juraj.stefanovic@paneurouni.com