# Implementation of a Fitness Function, Mutation Function and Elitism by Using Upgraded Petri Net

Jovo Arežina, Perica S. Štrbac
*Faculty of Computer Science*
Belgrade, Serbia
jovo.arezina@gmail.com strbac68@gmail.com

Zoran Banjac
*School of Electrical Engineering and Computer Science*
Belgrade, Serbia
zoran.banjac@viser.edu.rs

*Abstract* — **The objective of this paper is modeling, simulation and analysis of Upgraded Petri Net (UPN) model which implements a genetic algorithm. The UPN was developed for simulation and analysis of processes, particularly at the register transfer level. Original software for modeling and simulations of UPN, PeM (Petri Net Manager), is developed and used for all models described in this paper. This software supports: UPN formal theory, graphical modeling, simulation and analysis of an UPN model. This paper includes UPN theory, the UPN models of fitness function, mutation function, elitism, their simulation and analysis. The UPN models generate their results by execution of UPN. This execution is based on parallel firing of a group of transitions. The suitability of UPN for modeling of the systems based on a genetic algorithm is examined and established.**

***Key words — Upgraded Petri Net; Genetic Algorithm; Fitness Function; Mutation Function; Elitism; Modeling; Simulation; Analysis;***

## I. INTRODUCTION

Upgraded Petri nets are a formal mathematical apparatus which enables modeling, simulation and process analysis [1]. They enable interactive monitoring of process operations and its gradual improvement from the initial phase, all the way to the final version.

The hierarchical structure of an UPN gives wide possibilities for abstraction. This feature of the UPN provides the model implementation consisting at the same time of elaborate pieces essential for the analysis at a certain level, and also of some general pieces whose details are irrelevant for the analysis at the given level of abstraction [2].

This paper presents the usage of UPN in the example of modeling, simulation and analysis of a genetic algorithm [3] through its fitness function, mutation function and elitism.

The algorithm finds a given string using basic genetic algorithm principles. A goal string is set and an initial population of random strings is generated. Those are then mated and mutated until the goal string is matched. The mating is done by simply combining a half of a string from one parent and the other half from another. Mutation is then applied on an offspring so one randomly chosen character in the string is substituted by a randomly chosen character. Two parents generate two different offspring.

During a life cycle the population is sorted by fitness, a portion determined by the elitism factor is eliminated and the rest is mated in such a way that the most fit strings are given a chance to procreate first before the maximum population count is reached. The fitness is determined by the number of different characters in the corresponding positions of the goal string and a string being measured for fitness.

As an example, the first UPN model implements the mutation function, the second model implements the fitness function and the third model implements a cull function that removes items from the population based on elitism factor. For given input parameters the model generates results of these functions in a *.mem file whose content represents the analyzed system's memory state (image). Input parameters are determined by: initial marking, multiplicity of arcs, transition function, transition firing level, place attributes, and UPN conflict solving [1].

The UPN is an extension of an ordinary Petri Net and formal modeling tool appropriate for simulation and analysis of processes, particularly at the register transfer level (RTL). Unlike other classes of Petri Net, the UPN can be able to generate numerical results through its model execution at RTL level. This UPN model then can be used for analyzing hardware implementation of the model and also for checking given results.

## II. AN UPN FORMAL THEORY

Upgraded Petri nets formal theory is based on functions [1]. Upgraded Petri net is a 9-tuple:

$C = (P, T, F, B, \mu, \theta, TF, TFL, PAF)$

where:

$P=\{p_1, p_2, p_3, ..., p_n\}, n>0$
- a finite nonempty set of places $p_i$

$T=\{t_1, t_2, t_3, ..., t_m\}, m>0$
- a finite nonempty set of transitions $t_j$

$F: T\times P\rightarrow N_0$ - Input Function;
$B: T\times P\rightarrow N_0$ - Output Function;
$\mu: P\rightarrow N_0$ - Marking Function:
$\theta: T\times\Delta\rightarrow\lambda$ - Timing Function;
$TF: T\rightarrow A$ - Transition Function;

$TFL: T \rightarrow N_0$ - Transition Firing Level;
$PAF: P \rightarrow (x, y)$ - Place Attributes Function;

The input function assigns a non-negative number to an ordered pair $(t_i,p_j) \in T \times P$. The assigned non-negative number defines how many times the place pi is input as compared to the transition $t_i$. $N_0$ represents the set of non-negative integers. The set of places which are input as compared to the transition $t_j$ is presented as follows $*t_j = \{ p_i \in P, F(t_j, p_i) > 0 \}$. For the presentation of the place $p_i \in *t_j$ which have the standard input compared to the $t_j$, the sign $*t_j^S$ will be used, and sign $F^S(t_j, p_i)$ will be used for such input function. For the places $p_i \in *t_j$ with inhibitor input in relation to the $t_j$ transition, the sign $*t_j^I$ will be used and sign $F^I(t_j, p_i)$ for the input function.

The output function gives a non-negative integer to the ordered pair $(t_i,p_j)$. The assigned non-negative integer shows how many times the place $p_i$ is input in relation to the $t_i$ transition. The set of places which are input in relation to the $t_j$ transition is presented as follows $t_j* = \{ p_i \in P, B(t_j, p_i) > 0 \}$.

The marking function assigns a non-negative integer to the pi place. The marking function can be defined as n-dimension vector (marking): $\mu = (\mu_1, \mu_2, \ldots, \mu_n)$, where $n = |P|$. Instead of sign $\mu_i$ it can be used the sign $\mu(p_i)$.

The timing function $\theta$ assigns the probability $\lambda_{ij} \in [0,1]$ to an ordered pair $(t_i, j) \in T \times N_0$, i.e., $\lambda_{ij} = \theta(t_i, j)$.

The transition function gives an operation $\alpha_j \in A$ to the $t_j$ transition. Sign $A$ is the set of operations which can be assigned to the transition.

The firing level function of the transition gives a non-negative integer to the transition $t_j$. If this number not equals zero, it shows the number of $p_i \in *t_j$ places takes part in the transition firing, and if this number equals zero, then all the places $p_i \in *t_j$ affect the $t_j$ transition firing.

Place attributes function assigns an ordered pair $(x,y)$ to the place $p_i$. The $x$ component is a real number called $x$ attribute, and $y$ is a non-negative integer called $y$ attribute (i.e. $x \in R$, $y \in N_0$). Over the $x$ attribute belonging to the $p_i \in *t_j$ places, the $\alpha_j$ operation assigned to the $t_j$ transition executes where the order of operands in the operation $\alpha_j$ is defined by the $y$ attributes which belong to the $p_i$ place in accordance to $TFL$ function take part in the transition firing.

An Operation Assigned to a Transition: Function $TF$ assigns to a transition $t_j$ one operation. This operation can be: arithmetical operation, logical operation or file operation [1]. Inside the suite *PeM* a file which is a target of file operation function has an **.mem* extension. This **.mem* file is a text file and it is used for simulation of computer system memory. One line inside the **.mem* file refers to context of one memory location of computer system that we are modeling.

An arithmetical operation $\alpha_j \in A$, which is assigned to the transition $t_j \in T$, uses attributes $x$ which belong to the places $p_i \in *t_j$ as operands of that operation. A result of an arithmetical operation $\alpha_j \in A$ will be placed into the attributes $x$ which belong to the places $p_i \in t_j*$. The order of an operand (i.e. order of attributes $x$ which belong to the places $p_i \in *t_j$) in an arithmetical operation $\alpha_j \in A$ is defined by attributes $y$ which belong to the places $p_i \in *t_j$.

A logical operation $\alpha_j \in A$ which is assigned to the transition $t_j \in T$, uses attributes x which belong to the places $p_i \in *t_j$ as operands of that operation. If a result of the logical operation $\alpha_j \in A$ is logical false the transition $t_j \in T$ is disabled and will stay in that state until the result of this logical operation $\alpha_j \in A$ becomes logical true. The order of an operand (i.e. order of attributes $x$ which belong to the places $p_i \in *t_j$) in a logical operation $\alpha_j \in A$ is defined by attributes $y$ which belong to the places $p_i \in *t_j$.

A file operation $\alpha_j \in A$ which is assigned to the transition $t_j \in T$ performs over the context of a file which extension is equal to **.mem*. A File Operation $\alpha_j \in A$ addresses context of a **.mem* by using attribute $x$ which belongs to the places $p_i \in *t_j$. A result of this operation $\alpha_j \in A$ changes the value of attributes $x$ which belong to the places $p_i \in t_j*$. The result also can change attributes $y$ which belong to the places $p_i \in t_j*$, or can change context of addressed line into the **.mem* file.

An UPN Graph: Upgraded Petri Net is represented via formal mathematical apparatus or graphically. An UPN is represented by bipartite multigraph as is in Petri-net.

An Upgraded Petri Net executing represents change of system state from the current state to the next state. This migration from one state to the other one is triggered by firing of the transitions. By UPN executing: marking vector can be changed, contents of **.mem* file can be changed, and attributes which belong to the places $p_i \in t_j*$ of enabled transition $t_j$ can be changed.

A transition $t_j \in T$ can be enabled in Upgraded Petri Net: $C = (P, T, F, B, \mu, \theta, TF, TFL, PAF)$ if the next 3 conditions are satisfied:

1° If the timing function $\lambda_{jk} = \theta(t_j, k) > 0;$ (1)

2° If $TFL(t_j) > 0$ then $(\#p_i(S)) + (\#p_i(I)) = TFL(t_j)$, (2)
and if $TFL(t_j) = 0$ then $(\#p_i(S)) + (\#p_i(I)) = |*t_j|$,
where
$\#p_i(S)$ is a number of places $p_i \in *t^S_j$ such that
$\mu(p_i) \geq F^S(t_j, p_i)$, and
$\#p_i(I)$ represents a number of places $p_i \in *t^I_j$ for which
$\mu(p_i) = 0;$

3° If a logical operation $\alpha_j \in A$ assigned to the transition $t_j$, then the result of the operation $\alpha_j$ must be equal to true. (3)

A marking vector $\mu$ will be changed to new marking vector $\mu'$ by firing of transitions $t_j$, where:

$\mu'(p_i) = \mu(p_i) - F(t_j, p_i) + B(t_j, p_i)$, for places $p_i \in *t_j^S$
$\mu'(p_k) = \mu(p_k) + B(t_j, p_i)$, for places $p_k \in *t_j^I$

By firing of the transition $t_j$ an arithmetic operation is executed or a file operation is executed with respect to the operation that is assigned to $t_j$ by function $TF(t_j)$.

A logical operation which assigned to the transition tj by function $TF(t_j)$ will be executed if conditions (1) and (2) related to $t_j$ are equal to true.

A conflict in Upgraded Petri Net influences UPN executing. A conflict in UPN is the same as the conflict in Petri-net.

An UPN reachability tree graphically represents all possible marking vectors which can occurs during an UPN execution for given initial marking. Reachability tree shows all states which model can reach from the initial state. The UPN reachability tree is the same as the Petri Net reachability tree.

An UPN executing refers to a concurrent firing of the enabled. An UPN execution generates an UPN flammability tree. This tree is such tree where a node of the tree is a set of the transitions which are enabled at the same time. If there is the same node as the current node in the flammability tree then generating of the flammability tree will be stopped. There are four types of nodes in a flammability tree: root node, double node, dead node, and inner node.

## III. UPN MODELS OF A GENETIC ALGORITHM

UPN models representing three crucial functions of the genetic algorithm are described in this section. [3], [4], [5]

The first UPN model of a mutation function shown in **Error! Reference source not found.** mutates a string. Initial marking is $\mu_{(p1)} = \mu_{(p2)} = \mu_{(p8)} = \mu_{(p9)} = 1$. The initial X attribute of place *p-1* holds the address which contains length of a string and the initial X attribute of place *p-2* holds address which contains address of the string. The output arc from transition *t-2* to place *p-4* is doubled as the string address is used twice by the model. Transitions *t-1*, *t-2* and *t-6* become enabled.

Meanwhile transition *t-6* already chose a random character based on X attributes of places *p-8* and *p-9* which represent the range of printable ASCII characters. A randomly chosen character is now the value of place *p-10* X attribute.

In the end transition *t-7* is enabled and ready to store a randomly chosen character at a randomly chosen address within the original string. Thus the string is mutated.

The second UPN model of a fitness function shown in

UPN model of the fitness function (part 1)

and UPN model of the fitness function (part 2)

compares two stings of same length and stores the number of different characters at corresponding positions in each string. Initial marking is $\mu_{(p1)} = 1$.The initial attribute of place *p-2* holds the address of the iterator value. Place *p-13* X attribute holds the address of the first string, and attribute *p-14* holds the address of the second string. Place *p-19* attribute X

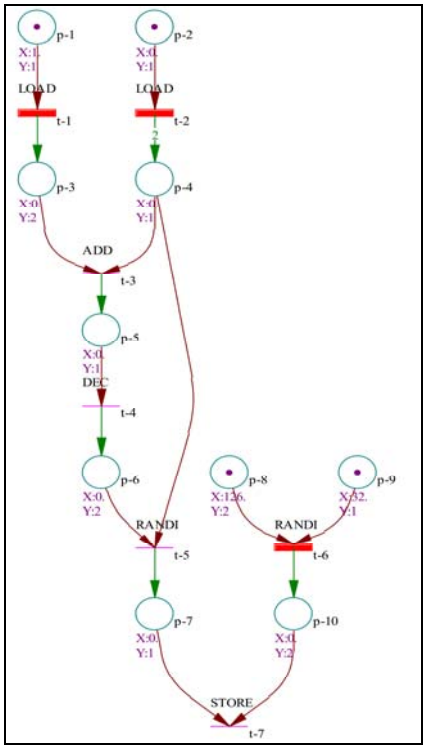


Figure 1. UPN model of the mutation function

holds the address of the fitness value. Transition *t-1* is enabled.

The address and length are loaded and then added together by transition *t-3*. After decrementing by transition *t-4* places *p-4* and *p-6* hold the addresses of the first and last characters in the string respectively. Transition *t-5* is able to randomly choose a character position in the string.

Meanwhile transition *t-6* already chose a random character based on X attributes of places *p-8* and *p-9* which represent the range of printable ASCII characters. A randomly chosen character is now the value of place *p-10* X attribute.

In the end transition *t-7* is enabled and ready to store a randomly chosen character at a randomly chosen address within the original string. Thus the string is mutated.

The second UPN model of a fitness function shown in

**UPN** model of the fitness function (part 1)

and **UPN model** of the fitness function (part 2)

compares two stings of same length and stores the number of different characters at corresponding positions in each string. Initial marking is $\mu_{(p1)} = 1$.The initial attribute of place *p-2* holds the address of the iterator value. Place *p-13* X attribute holds the address of the first string, and attribute *p-14* holds the address of the second string. Place *p-19* attribute X holds the address of the fitness value. Transition *t-1* is enabled.

A loop is created to decrement the iterator by outputting transition *t-6* to place *p-1*. Decremented value is stored back by transition *t-3*. If the iterator is greater than zero (X attribute of place *p-10*) the execution is allowed to continue. Otherwise the loop ends in transition *t-20* and no more markings are propagated through the loop.

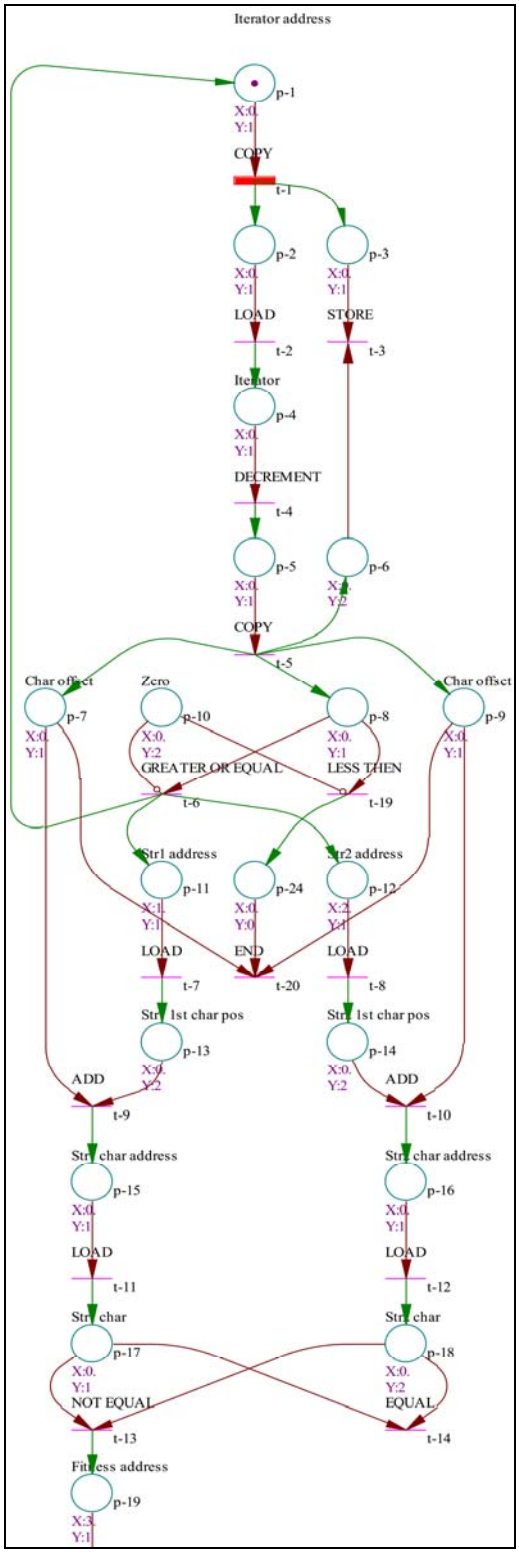


Figure 2. UPN model of the fitness function (part 1)

While the loop is running the iterator value is added to the address of both strings separately in transitions *t-9* and *t-10*. This results in addresses for the characters to be compared and they are loaded and compared by transitions *t-13* and *t-14*.

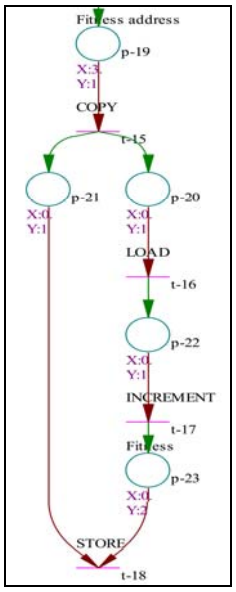


Figure 3. UPN model of the fitness function (part 2)

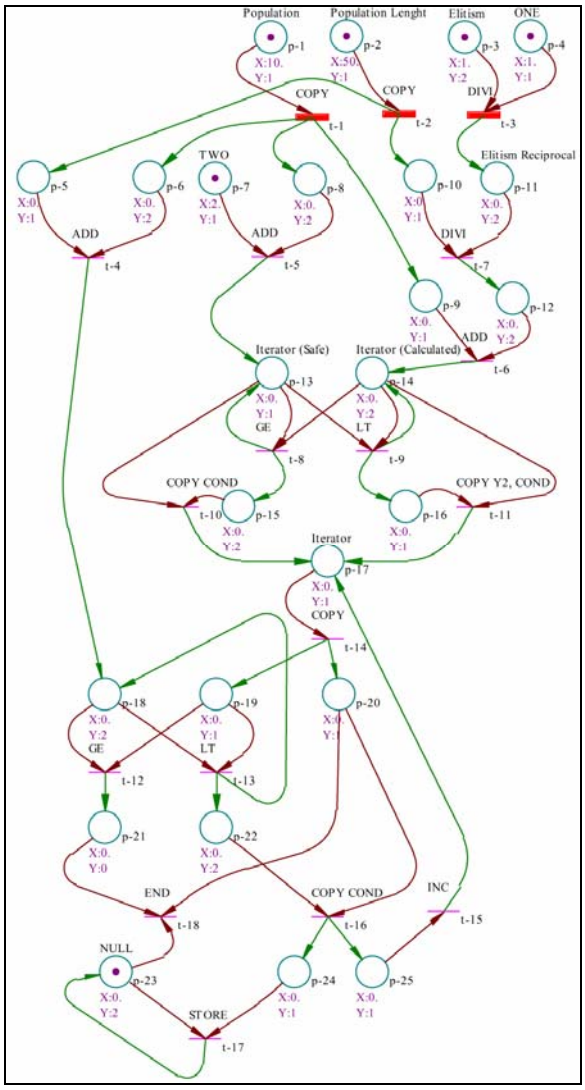


Figure 4. UPN model of the cull function

If the characters are not equal a branch that loads the current fitness value, increments it and stores it back is activated. In the end there are no markings in the model and a fitness value can be found at the designated memory register.

The third UPN model shown in **Fig. 4** performs a cull of the population based on the elitism factor [6], [7], [8]. In their X attributes places p-1, p-2 and p-3 hold the address of the first element of a sorted array that contains the population, the length of the array and the elitism factor, respectively. One branch of the model calculates the iterator position for the first array element to be made NULL (p-14), and the other branch sets the iterator to the third element in the array (p-13). The two iterator values are called Calculated and Safe and the larger of the two is chosen for the actual iterator (p-17). This is done to make sure that at least two members of the population survive a life cycle.

The model then proceeds to loop through the array making elements NULL until the iterator value gets greater then the address of the last element of the array.

## IV. AN UPN MODEL EXECUTION

By executing the UPN model for given initial marking shown on **Error! Reference source not found.** the following sequence of transitions firing will happen: *{t-1,t-2,t-6}→{t-3}→{t-4}→{t-5}→{t-7}*

By executing the UPN model for given initial marking shown on

UPN model of the fitness function (part 1)

and UPN model of the fitness function (part 2)

the next sequence of transitions firing will happen: *{t-1}→{t-2}→{t-4}→{t-5}*

After this sequence the next sequence depends of condition LT, GE *(t-6, t-19*) the execution could end in this sequence: *→{t-3, t-19}→{t-20}*
or continue:
*→{t-3, t-6}→{t-1, t-7, t-8}→{t-2, t-9, t-10}→{t-4, t-11, t-12}*

At this point the sequence depends on EQ and NEQ conditions (*t-13, t-14*). It can either end with {*t-5, t-14} or continue with {t-5, t-13}→{t-3, t-6, t-15}→{t-1, t-7, t-8, t-16}→ {t-2, t-9, t-10, t-17}→{t-4, t-11, t-12, t-18}*.

By executing the UPN model for initial markings shown on Fig. 4 the next sequence of transitions firing will happen: *{t-1, t-2, t-3}→{t-4, t-5, t-7}→{t-6}*. Then, based on the X attribute values of places *p-13* and *p-14* either *{t-8}→{t-10}* or *{t-9}→{t-11}* occurs. Transition *{t-14}* then fires and after that yet another branch is determined by the X attribute values of places *p-18* and *p-19* so either the sequence *{t-13}→{t-16}→{t-15, t-17}* or *{t-12}→{t-18}* will occur.

## V. THE UPN MODEL ANALYSIS

The results given by the UPN models are expected and can be checked by comparing *.mem files before and after execution. *Mutate.mem* most probably shows that a change occurred in the string (if the same character was not chosen) and *Fitness.mem* shows the calculated fitness value in the expected register. *Cull.mem* shows the population array with NULL-ed elements if the elitism factor was set appropriately.

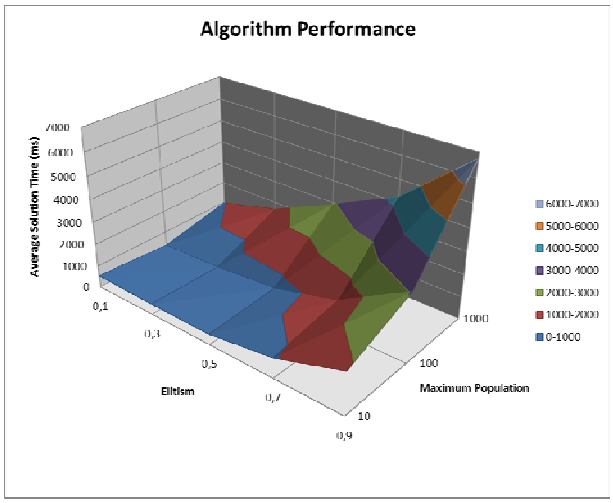


Figure 5. Algorithm performance data

Furthermore a great potential for parallel operation is observed and implemented in software using *C#* language, although a deterministic hardware implementation would benefit from parallelism even more.

**Fig. 5** shows results obtained with different elitism and maximum population settings. Every data point is an average of 40 runs on a goal string with the length of 44. The variable of a randomly chosen initial population is avoided by using the same generated population for each run of an input combination, but the variables of computer hardware performance and random mutations remain. This prevents the usage of a different aggregation function for data points (e.g. minimum).

The data shows a clear advantage in using aggressive elitism settings and a small population. This in turn implies that the cost of operating a large population outweighs the benefits of increased probability of reaching the goal string. The benefits of aggressive elitism settings are expected based on the algorithm as they ensure the occurrence of more mutations.

## VI. Conclusion

UPN are suitable for implementing a genetic algorithm at the register transfer level (RTL). Two models are presented. The first model is the mutation function. The second UPN model is the fitness function. This method is chosen as an example for implementation. Given UPN models can generate results for given input data and initial marking. By executing of given UPN models the net passes through many states. Finally, when a fitness value is found, a string is mutated, or the population is culled execution terminates. Suitability of given UPN model was checked by execution of the model and the results generated dynamically during this execution. UPN can be used for creating models based on genetic algorithms due to their parallel nature and the need for random inputs. Unlike other classes of Petri net the UPN can be able to generate numerical results through its model execution at RTL level. This level is suitable for analyzing hardware implementation of the model and also for checking given results. Original software for modeling and simulations of Upgraded Petri Nets, PeM (Petri Net Manager), is developed and used for all models described in this paper.

## Acknowledgements

This research was supported by the Serbian Ministry of Education and Science, Republic of Serbia, Project no. III 44006

## References

[1] Perica Strbac, Milan Tuba, “An Upgraded Petri Net Model, Simulation and Analysis of An 8x8 Sub-Image for JPEG Image Compression,” chapter in the book Recent Advances in Applied Informatics and Communications, Recent Advances in Computer Engineering, WSEAS Press, ISBN 978-960-474-107-6, Moscow, Russia, pp. 167-172, August 2009.

[2] J. L. Peterson, “Petri Net Theory and Modeling of Systems”, Englewood Cliffs , Prentice Hall, New York 1981.

[3] K. F. Man, K. S. Tang and S. Kwong, “Genetic Algorithms: Concepts and Applications,” IEEE Transactions on Industrial Electronics, Vol. 43, No. 5, pp. 519-534, October 1996.

[4] Chang Wook Ahn and R. S. Ramakrishna, “Elitism-Based Compact Genetic Algorithms,” IEEE Transactions on Evolutionary Computation, Vol. 7, No. 4, pp. 367-385, August 2003.

[5] D. Dumitrescu, B. Lazzerini, L. C. Jain, and A. Dumitrescu, “Evolutionary Computation,” Boca Raton, FL: CRC Press, 2000.

[6] S. Yang, “Genetic Algorithms with Elitism-based Immigrants for Changing Optimization Problems,” Lecture Notes in Computer Science, Volume 4448/2007, pp. 627-636, DOI: 10.1007/978-3-540-71805-5_69, 2007.

[7] S. Yang, “Genetic algorithms with memory- and elitism-based immigrants in dynamic environments,” Evol. Comput., vol. 16, no. 3, pp. 385–416, Sept. 2008.

[8] Mishra K.K., Tiwari S., Kumar A., Misra A.K., “An approach for mutation testing using elitist genetic algorithm,” International Conference on Computer Science and Information Technology 3rd IEEE (ICCSIT), pp. 426-429, 2010.