

Chapter 3

Methodology

This chapter explains how to represent the problem on computing unique input output (UIO) sequences of sample instances of finite state machine and how to design the fitness function for find the solution. There is the new hypothesis for investigating in this project. The Genetic Algorithm and Self-Adaptive Evolutionary Algorithm which are applied to solving problem on computing UIO sequences of a sample instances of finite state machines are displayed in this chapter. These algorithms may also compute UIO sequences of real-world instances of finite state machines. In addition, there are the explanation how to compare and analyse the performance of both algorithms.

Hypothesis: The Self-Adaptive Evolutionary Algorithm should be faster in the performance of computing UIO sequences of finite state machines than the Genetic Algorithm.

3.1 The representation of the problem

The problem on computing UIO sequences for conformance testing of finite state machines is the critical problem because in software testing, every software systems have to verify a quality and performance. However, the computing UIO sequences of finite state machines are sometimes difficult to predict correct UIO sequences as expected. We will therefore consider on computing UIO sequences in the different instances of both sample and real-world finite state machines by using The Genetic Algorithm and Self-Adaptive Evolutionary Algorithm.

In this project, a sample finite state machine is designed as the representation of the problem. The simple finite state machine contains some states and transitions which provide some specific behaviours. There are input and output symbols in all states and transitions which identify behaviours. The simple finite state machine is $M = (I, O, S, \delta, \lambda)$. The definition of designing the simple finite state machine is as follows:

- The number of states = 4
- The number of input symbols = 3
- The length of input sequences = $\{1, 2, 3, 4, \dots, n\}$
- The set of input symbols: $I = \{a, b, c\}$

- The set of out symbols: $O = \{0, 1, 2, 3\}$
- The set of states: $S = \{s0, s1, s2, s3\}$
- Transition function $\delta: \{s \times I \rightarrow S \mid s \in S\}$
; e.g. $s0 \times a \rightarrow s0$, $s0 \times b \rightarrow s1$, $s0 \times c \rightarrow s2$.
- Output function $\lambda: \{s \times I \rightarrow O \mid s \in S\}$
; e.g. $s0 \times a \rightarrow 0$, $s0 \times b \rightarrow 2$, $s0 \times c \rightarrow 1$.

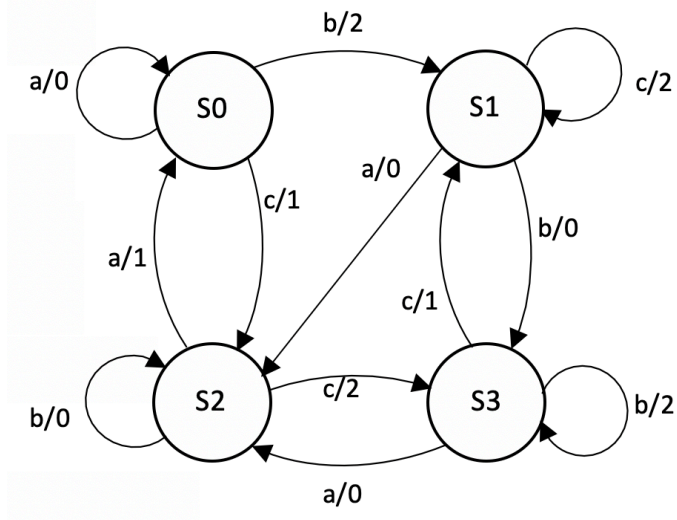


Figure 3.1 The simple finite state machine.

start state	input/output	end state
s0	a/0	s0
	b/2	s1
	c/1	s2
s1	a/0	s2
	b/0	s3
	c/2	s1
s2	a/1	s0
	b/0	s2
	c/2	s3
s3	a/0	s2
	b/2	s3
	c/1	s1

Table 3.1 The transition table for the simple finite state machine.

The simple finite state machine and transition table illustrate in Figure 3.1 and Table 3.1 in respectively. In fundamental work, when a certain input is read on a start of the finite state machine, the machine will move to the next state and provide the certain output by computing from transition function and output function. In this machine, the state s_0 is defined as the start state. When any certain inputs are read as the start state on the machine, the machine may produce certain outputs and transitions to the end state.

However, one certain input may not provide the specific characteristic for each state of this finite state machine. In each state, one input sequence for a state will obtain the specific characteristic which can be called a UIO sequence of the state. Sometimes, the length of any input sequences may be different. Hence, the input sequence of each state which will be UIO sequence must be difference.

In the definition of UIO sequence, we consider in the pair of the transitions and focus on the output of the states. For example, state s and state t are considered in the UIO sequence, the definition is $\lambda(s \times a) \neq \lambda(t \times a)$; where an input $a \in I$ and for all states t , $t \neq s$. The same input is read on the machine at state s and t , then the output of state s and state t must be different. When the state s is compared to all state t and the output of all state t , there are no same output as state s . It means that the input 'a' is a unique input output of the state s . An example of UIO sequence of state s_1 is considered as follows:

Example: we suppose an input sequence as 'acb'. the output of each state will be below.

s_0 : '010'

s_1 : '022'

s_2 : '110'

s_3 : '022'

When we consider in state s_0 , 'acb' is an input sequence of this finite state machine, the output for this input sequence is '010'. There are no same outputs between s_0 and other states. Therefore, 'acb' is a UIO sequence of state s_0 because in the same input, there is no same output in other states.

The way to represent the problem on computing UIO sequences is to seek the possible input sequences of the finite state machine. All possible input sequences which are the candidate solutions are generated by a program. These possible input sequences are computed the certain outputs and transitions based on the finite state machine.

3.1.1 The possible input sequences encoding

On this work, the problems are randomly input sequences which are generated by UIO generator. There are two programs for computing UIO sequences which are the Genetic Algorithm (the GA) and Self-Adaptive Evolutionary Algorithm (the Self-Adaptive EA). UIO sequences are as the possible problems which can be called ‘an individual’. There are three symbols which are ‘a’, ‘b’ and ‘c’, they will be encoded to the numbers which are ‘0’, ‘1’ and ‘2’ respectively. For example, in figure 3.1, the individuals are shown in Table 3.2.

Start states	Input Sequences	encoded (individuals)	Output Sequences	Check UIO of s1	End states
s0	‘aaaa’	‘0000’	‘0000’	‘aaaa’/ ‘0000’ is a UIO of s1	s0
s1	‘aaaa’	‘0000’	‘0100’		s0
s2	‘aaaa’	‘0000’	‘1000’		s0
s3	‘aaaa’	‘0000’	‘0100’		s0
s0	‘b’	‘1’	‘2’	‘b’/ ‘1’ is NOT a UIO of s1	s1
s1	‘b’	‘1’	‘0’		s3
s2	‘b’	‘1’	‘0’		s2
s3	‘b’	‘1’	‘2’		s3
...

Table 3.2 The example of input/output sequences for each state on the machine.

For the hypothesis, it notes that the Self-Adaptive EA should be more effective than the Genetic Algorithm when considering on the computing UIO sequences of the finite state machine. The objective of the project will follow to investigate this hypothesis. The fitness function for seeking the optimal solution of computing UIO sequences will be presented in the next part.

3.2 The fitness function

The fitness function (1) for computing UIO sequences for conformance testing of the simple finite state machine M is defined as follows:

$$F_{M,s}(x) = n - \gamma_M(s, x) \text{ where } \gamma_M(s, x) = |\{t \in S \mid \lambda(s, x) = \lambda(t, a)\}| \quad (1)$$

n is the number of the states of the finite state machine M .

$\gamma_M(s, x)$ is the number of the states at leaf node.

The average function (2) is an average of the fitness results which will be used to analyse the different distributed results between the GA and the Self-Adaptive EA. The average A is derived from the fitness function $F_{M,s}(1)$. The function is shown below.

$$A = \frac{\sum_{i=1}^m F_{M,s}(x)}{m} \quad (2)$$

A is an average of the fitness values.

m is the number of iterations.

$F_{M,s}$ is the fitness result of computing UIO sequences.

For example of considering on the fitness function (1), to suppose, the input sequence is 'acb' and observe on the state s_0 . The partition tree can be illustrated following the finite state machine in Figure 3.2. It can be seen that when we consider state s_0 with the input sequence as 'acb', the program will find the output based on the input sequence and check that it is the UIO of state s_0 or not. In Figure 3.2, at the leaf node, only state s_0 is in the first group, $\gamma_M(s_0, 'acb')$ will be 1. Therefore, to calculate by fitness function, the fitness value is $F_{M,s_0}('acb') = 4 - 1 = 3$. Additionally, if we consider on state s_1 ; $F_{M,s_1}('acb') = 4 - \gamma_M(s_1, 'acb')$, the fitness value of this example is $F_{M,s_1}('acb') = 4 - 2 = 2$.

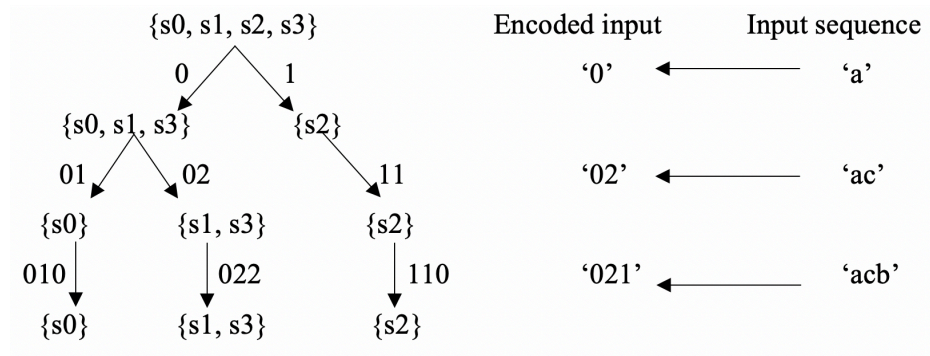


Figure 3.2 The partition tree for the input sequence 'acb' in the finite state machine.

Furthermore, there are some pseudo-codes for the fitness function which is implemented on the Genetic Algorithm and Self-Adaptive Evolutionary Algorithm. The pseudo-code of the fitness function will display below.

The fitness function (CalculateUIOFitness):

Require: *the set of outputs (fsmOutputSet), the set of transitions (fsmStateTransitionSet), inputSeq, NumState*

Each individual

fitness \leftarrow NumState - **transitionMany** (fsmOutputSet, fsmStateTransitionSet, inputSeq)
return fitness

TransitionMany:

Require: *the set of outputs (fsmOutputSet), the set of transitions (fsmStateTransitionSet), inputSeq, NumState, m; where m is the number of the stages, n; where n is length of individual.*

Each individual

Reset(); // reset the state.
for each state s_i , $i = 1$ to m ; **do**
 {
 for each bit-string $j = 0$ to n **do**
 {
 if ($j \neq 0$) then {
 CurrentState \leftarrow NextState
 }
 //compute an output of each bit-string.
 output[j] = Transition (...);
 //update state
 NextState = getCurrentState (...);
 } **end for**
 count how many time other outputs will be same with the output (s1)
 } **end for**
 return count (*the number of leaf node include s1*).

3.3 The Genetic Algorithm (the GA)

The GA is applied to compute the possible UIOs of the simple finite state machine. The GA's processes will be explained below. There are four main processes in the GA which are Selection, Crossover, Mutation and Evaluation, as shown below.

1. In Selection process, two parents or two individuals are chosen at random by tournament selection for generating new offspring by Crossover and Mutation.
2. Then, two parents will be crossover to get a new offspring. A crossover point will be set at random between two parents to get the new offspring.

An example; '001|102'; 'aab|bac' (1st parent with crossover point at 0.5)

'001|111'; 'aab|bbb' (2nd parent with crossover point at 0.5)

'001|111'; 'aab|bbb' (a new offspring)

3. The Mutation process will change some bit-string on the new offspring. The amount of the probability of mutation will be as $1/n$ where n is a length of the individual (offspring).

An example; '001|111'; 'aab|bbb' (a new offspring)

'000|111'; 'aaa|bbb' (a mutated offspring)

4. The last process is the Evaluation process which will evaluate and compute the individuals. The individuals will be reproduced for the next population for the next generation (iteration). The pseudo-code of the GA is also shown as follows.

The GA:

Require: the fitness function, initial population P_0 , n = the length of the individual.

Initialisation (population P_t)

for $t = 0, 1, 2, \dots$ to end condition met **do**

{

Evaluation (P_t) by using 'CalculateUIOFitness' // the fitness function (1)

for $i=0 \dots \lambda$ **do**

{

Choose parents (x, y) by tournament selection

Offspring $x \leftarrow$ by crossover x and y with probability $1/2$ of individuals

Get Offspring $x' \leftarrow$ by flipping each bit-string of x with probability $1/n$

} **end for**

```

//update  $P_{t+1}(i)$ 
Set population  $P_{t+1}(i) = (x')$ 
Aver  $\leftarrow$  Record the fitness value of each iteration
} end for
Average = Aver/iterations // the average function (2)
//end the program

```

The diagram of a basic work of the Self-Adaptive EA represents in Figure 3.3.

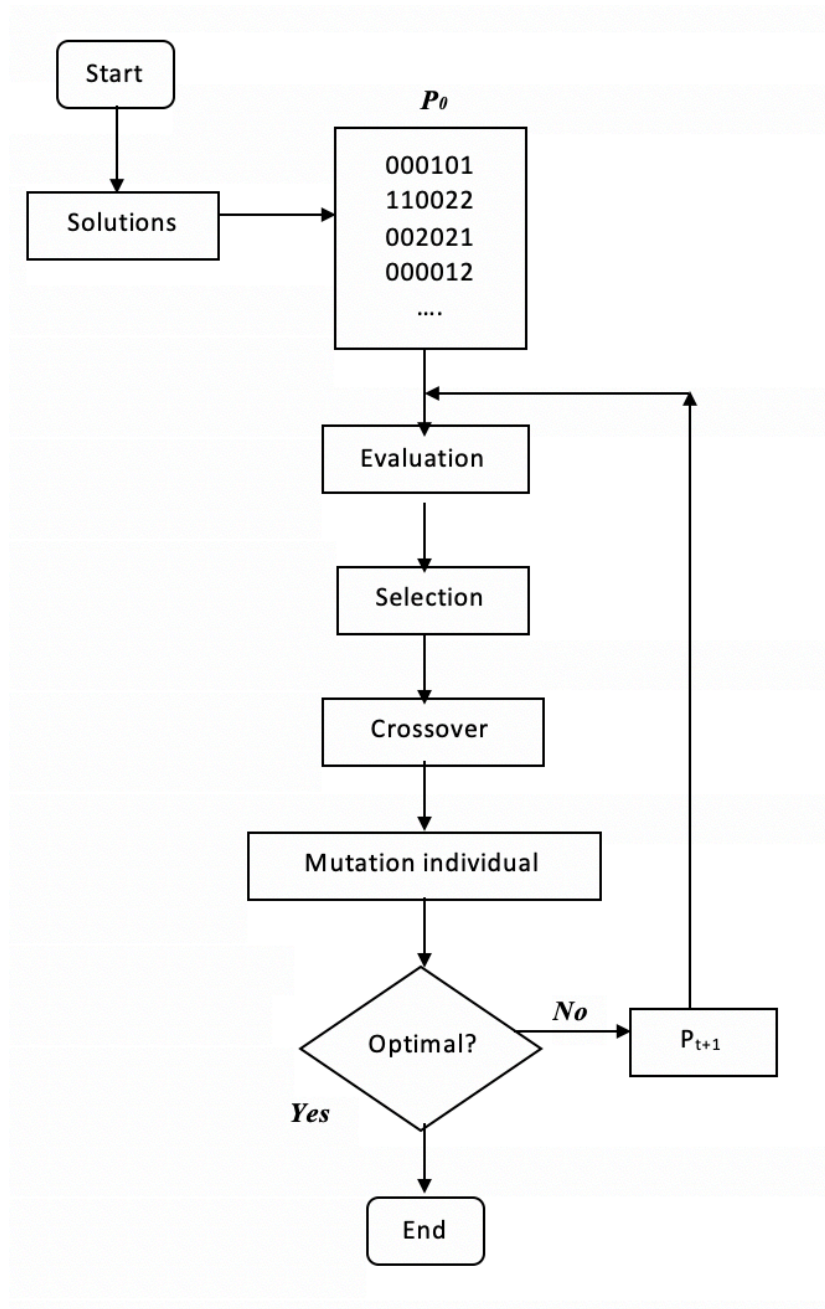


Figure 3.3 The diagram of the GA.

3.4 The Self-Adaptive Evolutionary Algorithm (the Self-Adaptive EA)

The Self-Adaptive EA is used to seek the UIO sequences of the finite state machine. The main concept of the Self-Adaptive EA is involved with two mutations. The phases of the Self-Adaptive EA are Selection, Mutation rate, Mutation and Evaluation. The description is presented as follows.

1. The Selection will select the individual x which has the best fitness value by sorting.
2. In the Mutation rate, there is the random rate (x_r) which set by user manual, then the random rate will be mutated to a new rate ($x_r \rightarrow x_r'$) before start to the next step.
5. Then, the individual will be mutated to new offspring based on the probability of $\frac{\text{Mutation Rate } (x_r')}{n}$; where n is the number of the bit-strings of the individual.
6. The Evaluation process will evaluate and compute the individuals. The individuals will be reproduced to the next generation. The pseudo-code of the Self-Adaptive EA is shown as follows:

The Self-Adaptive EA:

Require: the fitness function, initial population P_0 , n = the length of the individual.

Initialisation (population P_t)

for $t = 0, 1, 2, \dots$ to end condition met **do**

{

Evaluation (P_t) by using 'CalculateUIOFitness' // the fitness function (1)

Sort P_t ; $P_t(0) \geq P_t(1) \geq P_t(\lambda)$

for $i=0 \dots \lambda$ **do**

{

Choose individual x (parent) and set the rate $X(x, X)$

if ($X > 1.1$) **then** {

$X' = n/2$

} **else** $X' = \varepsilon(1/10)$

Get $x' \leftarrow x$ by flipping each bit-string of x with probability X'/n

} **end for**

//update $P_{t+1}(i)$

Set population $P_{t+1}(i) = (x', X')$

Aver \leftarrow Record the fitness value of each iteration

} end for

Average = Aver/iterations

// the average function (2)

//end the program

The diagram of a basic work of the Self-Adaptive EA represents in Figure 3.4.

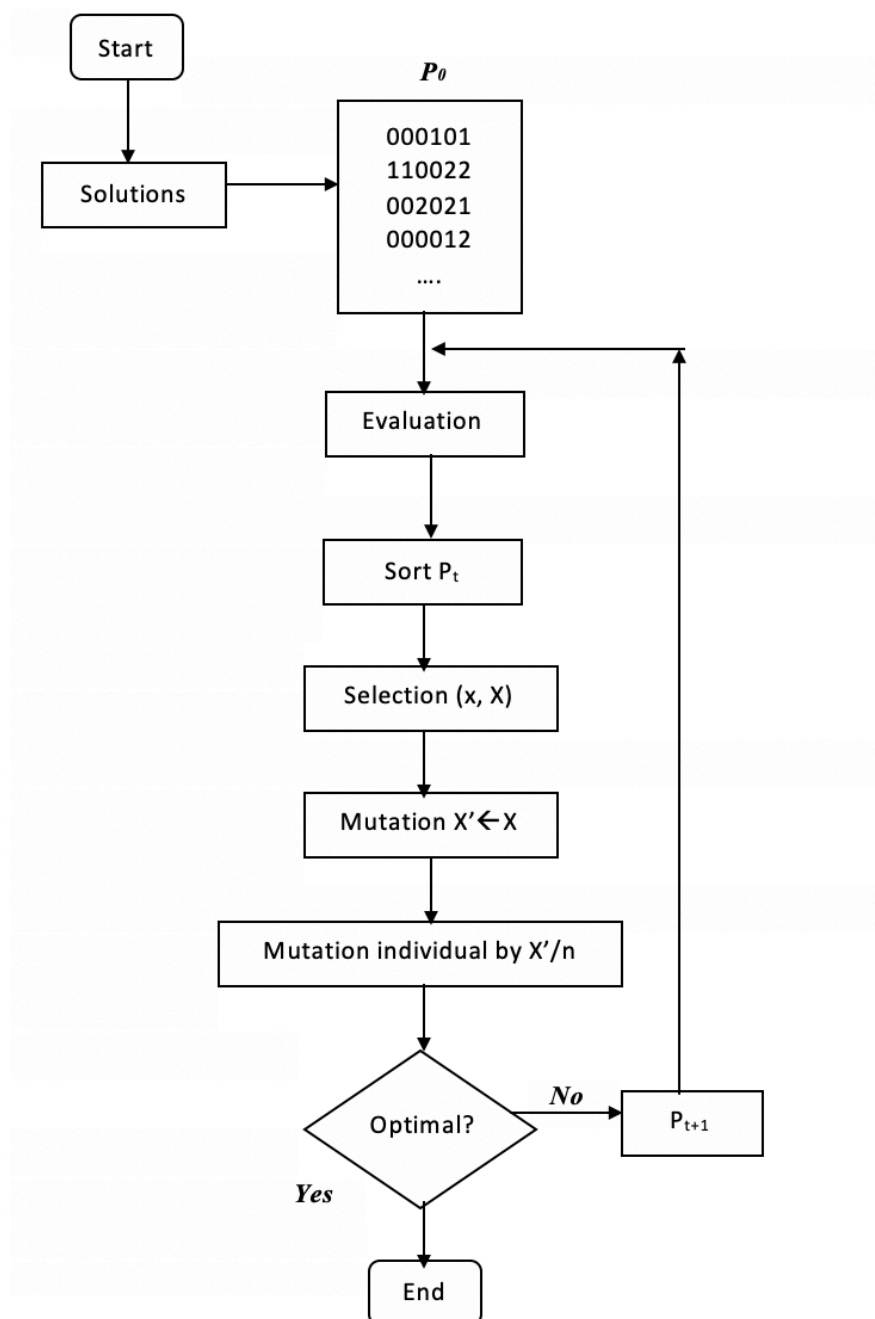


Figure 3.4 Diagram of The Self-Adaptive EA

3.5 The analysis on the performance of both algorithms

To analyse the performance between the GA and the Self-Adaptive EA, there is monitoring and recoding the collection of the results on computing unique input output (UIO) sequences of the finite state machine. The results of both programs will be compared on the performance which algorithm is better to find UIO sequences of the finite state machine. This analysis will also be the evidence to support the hypothesis that “the Self-Adaptive EA could be more effective to find UIO sequences of the finite state machine than the GA.”

A test system which is Wilcoxon Rank Sum Test or Mann Whitney U is used for analysis the difference of the distribution results of both programs (Wayne W., 2017). This test is nonparametric test for checking the equality or difference between the results of two samples (Wayne W., 2017).

We therefore use the Wilcoxon Rank Sum Test for analysing the results of the computing unique input output (UIO) sequences of the finite state machine. The Wilcoxon Rank Sum Test will observe the value of distribution of the results between two programs. It can provide which program is effective for computing unique input output (UIO) sequences based on the results of the programs. Then, the result of the Wilcoxon Rank Sum Test also considers to the hypothesis. In the result of this test, it will calculate a value from two different data of two programs which can call ‘p-value’. “R application” is used for computing the p-value for analysis the performance of both programs.

3.6 The software testing

In software testing, unit testing and system testing will be used to check the validity of work of the programs. The unit testing is a small testing which will check each method or process of the program while the system testing is a test which has been used for checking whole program.

3.6.1 The unit testing for FSM

The unit testing of a part of the program for testing the simple finite state machine is shown in Figure 3.5. We identify the simple finite state machine (Figure 3.1), start state (i.e. s1 encoded to 0) and simple input sequence (i.e. ‘acb’ encoded to ‘021’). The program will check initial state, then check the output by using the input sequence and update the next state. The program

will work until the input sequence is checked every bit. The output sequence of the input sequence '021' is presented in Figure 3.5.

The screenshot shows an IDE with two tabs: 'GA_Demo.java' and 'fsm.java'. The 'Source' tab is active, displaying the following Java code:

```

18     int currentState=0;
19
20     public int[] transitionMany (int[][] fsmOutputSet,int[][] fsmNextState, int[] inputSeq){
21         int[] output = new int[inputSeq.length];
22         reset();
23         System.out.println("Start State : s1 / "+ startState);
24         System.out.println("InputSeq : "+Arrays.toString(inputSeq));
25         for(int i=0; i<inputSeq.length; i++){
26             //check each state
27             if(i!=0){
28                 currentState = nextState;
29             }
30             output[i]= transition(fsmOutputSet,currentState,inputSeq[i]);
31             nextState = getCurrentState(fsmNextState,currentState,inputSeq[i]);
32             //update state
33             System.out.println("current state :"+ currentState + "    Output :"+output[i] + "
34         }
35         System.out.println("OutputSeq : "+Arrays.toString(output));
36         return output;
37     }
38
39     //reset the machine

```

Below the code editor, the 'Output - GA_Demo (run)' window shows the following output:

```

run:
Start State : s1 / 0
InputSeq : [0, 2, 1]
current state :0    Output :0    Next State :0
current state :0    Output :1    Next State :2
current state :2    Output :0    Next State :2
OutputSeq : [0, 1, 0]
BUILD SUCCESSFUL (total time: 0 seconds)

```

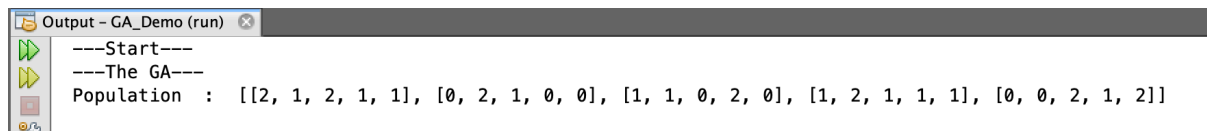
Figure 3.5 The unit testing for the simple finite state machine M.

3.6.2 The unit testing for computing UIO and the fitness function

The unit testing is also used to verify the process of computing UIO sequences of the finite state machine. After computing the output sequences in each state, this test will check the correction of comparing the output sequences. On state s1, if there are no same output sequences when considering the same input sequence with other states, the result of the program should present that “this input sequence is a UIO for state s1”. This test will run for observing the computation of each state, the output sequences of each state and the comparison of every states. The unit testing of computing UIO and the fitness function is shown in Figure 3.6 and 3.7.

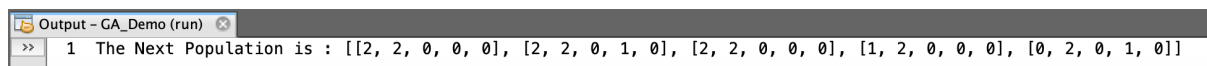
3.6.4 The unit testing of the processes in the GA

The GA will generate the initial population (P) at random based on the number of the population, the member of input symbols (i.e. a, b, c) and the length of each individual. At the beginning, the GA will generate the initial population automatically, the unit testing for this work is presented in Figure 3.9. Figure 3.10 and Figure 3.11 show the next population (P+1) and (P+9) of the individuals.



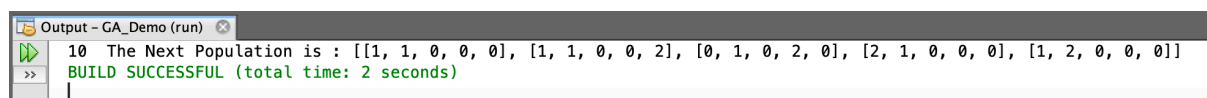
```
Output - GA_Demo (run)
---Start---
---The GA---
Population : [[2, 1, 2, 1, 1], [0, 2, 1, 0, 0], [1, 1, 0, 2, 0], [1, 2, 1, 1, 1], [0, 0, 2, 1, 2]]
```

Figure 3.9 The unit testing for creating population (length = 5 and $P = 5$ individuals).



```
Output - GA_Demo (run)
>> 1 The Next Population is : [[2, 2, 0, 0, 0], [2, 2, 0, 1, 0], [2, 2, 0, 0, 0], [1, 2, 0, 0, 0], [0, 2, 0, 1, 0]]
```

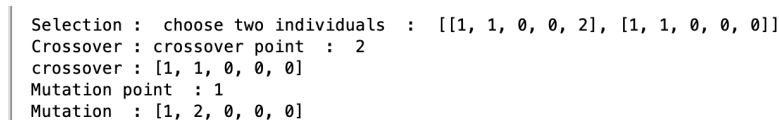
Figure 3.10 The unit testing for creating population (length = 5 and $P+1 = 5$ individuals).



```
Output - GA_Demo (run)
>> 10 The Next Population is : [[1, 1, 0, 0, 0], [1, 1, 0, 0, 2], [0, 1, 0, 2, 0], [2, 1, 0, 0, 0], [1, 2, 0, 0, 0]]
BUILD SUCCESSFUL (total time: 2 seconds)
```

Figure 3.11 The unit testing for creating population (length = 5 and $P+9 = 5$ individuals).

The selection process chooses two individuals as parents from the population and then crossover the parents to obtain the offspring. After that go forward to the mutation process, the offspring will be mutated by probability of $1/n$ where n is the length of input sequences. The unit testing of selection, crossover and mutation processes is shown in Figure 3.12.



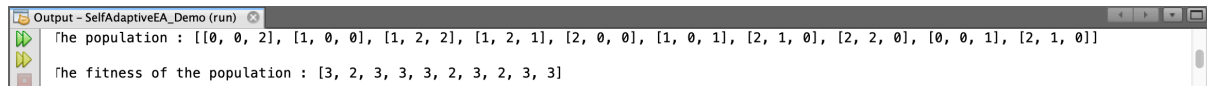
```
Selection : choose two individuals : [[1, 1, 0, 0, 2], [1, 1, 0, 0, 0]]
Crossover : crossover point : 2
crossover : [1, 1, 0, 0, 0]
Mutation point : 1
Mutation : [1, 2, 0, 0, 0]
```

Figure 3.12 The unit testing for selection, crossover and mutation processes in an iteration.

3.6.5 The unit testing of the processes the Self-Adaptive EA

The unit testing for checking the processes of the Self-Adaptive EA is quite similar the GA. However, there is the unit testing for the comparison of the fitness values of the population in

the Self-Adaptive EA. This comparison will sort the population from the best fitness value to the worst fitness value. The best individual which contains the fitness value will be chosen in the Selection process. For example, we suppose 10 individuals (the population) which contain three bit-strings in each one for checking the sorting of the individuals. Figure 3.13 presents the population and the fitness values of the population and Figure 3.14 illustrates the order of population after sorting and comparing.

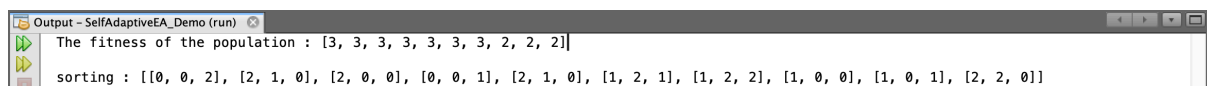


```

The population : [[0, 0, 2], [1, 0, 0], [1, 2, 2], [1, 2, 1], [2, 0, 0], [1, 0, 1], [2, 1, 0], [2, 2, 0], [0, 0, 1], [2, 1, 0]]
The fitness of the population : [3, 2, 3, 3, 3, 2, 3, 2, 3, 3]

```

Figure 3.13 The population before comparing and sorting.



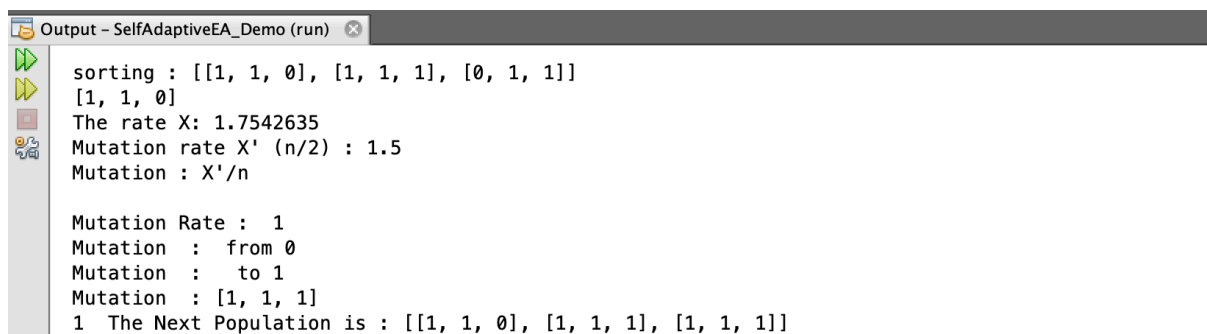
```

The fitness of the population : [3, 3, 3, 3, 3, 3, 2, 2, 2, 2]
sorting : [[0, 0, 2], [2, 1, 0], [2, 0, 0], [0, 0, 1], [2, 1, 0], [1, 2, 1], [1, 2, 2], [1, 0, 0], [1, 0, 1], [2, 2, 0]]

```

Figure 3.14 The unit testing for sorting population.

Furthermore, the Self-Adaptive EA has some processes that are difference from the GA. There are two processes which are Mutation rate process and Mutation process. Two processes are checked by the unit testing. The unit testing of the Mutation rate process and Mutation process are shown in Figure 3.15.



```

sorting : [[1, 1, 0], [1, 1, 1], [0, 1, 1]]
[1, 1, 0]
The rate X: 1.7542635
Mutation rate X' (n/2) : 1.5
Mutation : X'/n

Mutation Rate : 1
Mutation : from 0
Mutation : to 1
Mutation : [1, 1, 1]
1 The Next Population is : [[1, 1, 0], [1, 1, 1], [1, 1, 1]]

```

Figure 3.15 The unit testing for the Mutation rate and Mutation process.