# Computing Unique Input/Output Sequences Using Genetic Algorithms

Qiang Guo, Robert M. Hierons, Mark Harman, and Karnig Derderian
Department of Information System and Computing
Brunel University, UB8 3PH, UK

## Abstract

The problem of computing Unique Input/Ouput sequences(UIOs) is NP-hard. Genetic algorithms (GAs) have been proven to be effective in providing good solutions for some NP-hard problems. In this work, we investigated the construction of UIOs using GAs. We defined a fitness function to guide the search of potential UIOs and introduce a **DO NOT CARE** character to improve the GA's diversity. Experimental results suggest that, in a small system, the performance of the GA based approaches is no worse than that of random search while, in a more complex system, the GA based approaches outperform random search.

***Keywords:*** *FSMs, UIOs, Conformance Testing, Genetic Algorithms, Optimisation*

## 1 Introduction

Finite state machines (FSMs) have been used for modelling systems in various areas such as sequential circuits, software and communication protocols [1, 2, 4, 9, 10, 11, 12]. Four test sequence generation methods are discussed and compared in [4], namely, Transition Tours (T-Method), Unique Input/Output Sequence (U-Method), Distinguishing Sequence (D-Method), and Characterizing Set (W-Method). The last three methods are known as formal methods since they not only check the transitions, but also verify the states. In terms of the fault coverage, the U-, D-, and W-Methods achieve better performance than T-Method does, while exhibiting no significant difference among themselves [4].

Among the formal methods, U-Method is popular since it benefits from the facts: (1) Not all FSMs have a Distinguishing Sequence (DS), but nearly all FSMs have UIOs for each state [6]; (2) The length of a UIO is no longer than DS; (3) While UIOs may be longer than a characterising set, in practice UIOs often lead to shorter test sequences. Unfortunately, computing UIOs is NP-hard [2]. Lee *et al.* [2] note that an adaptive distinguishing sequences and UIOs may be produced by constructing a state splitting tree. However, no rule is explicitly defined to guide the construction of input sequence. Naik [5] proposes an approach to construct UIOs by introducing a set of inference rules. Some minimal length UIOs are found. These are used to deduce some other states' UIOs. A state's UIO is produced by concatenating a sequence to another state, whose UIO has been found, with this state's UIO sequence. Although it reduces computational complexity, the inference rule inevitably increases a UIO's length, which will consequently add more costs to the forthcoming test.

Genetic algorithms (GAs) have proven efficient in search and optimisation [7] and have shown their effectiveness in providing good solutions to NP-hard problems such as the Travelling Salesman Problem. This work investigates the use of GAs for constructing UIOs from an FSM. An initial population is produced by randomly generating input sequences. This population is used to explore potential UIOs. Based on the state splitting tree, a fitness function is defined to evaluate the quality of the input sequences. This fitness function encourages candidates to split the set of all states into more discrete units and punishes the length of the sequences. Roulette wheel selection and uniform crossover are implemented. Simulation results are also presented and discussed. During the evolutionary computation, good solutions found are stored in a database. This database can be used to preserve the information

lost during the computation and to further optimise UIOs' length.

This paper is organised as follows: FSMs are briefly reviewed in section 2. A simple GA is introduced in section 3. Experiments and corresponding results are described in section 4. Finally, conclusions are drawn in section 5.

## 2 Preliminaries

### 2.1 Finite State Machines

An FSM $M$ is defined as a quintuple $(I,O,S,\delta,\lambda)$ where $I,O,$ and $S$ are finite and nonempty sets of input symbols, output symbols, and states, respectively; $\delta : S \times I \longrightarrow S$ is the state transition function; and $\lambda : S \times I \longrightarrow O$ is the output function. When the machine is in a current state $s \in S$ and receives an input $a \in I$, it moves to the next state $\delta(s,a)$ and produces output $\lambda(s,a)$.

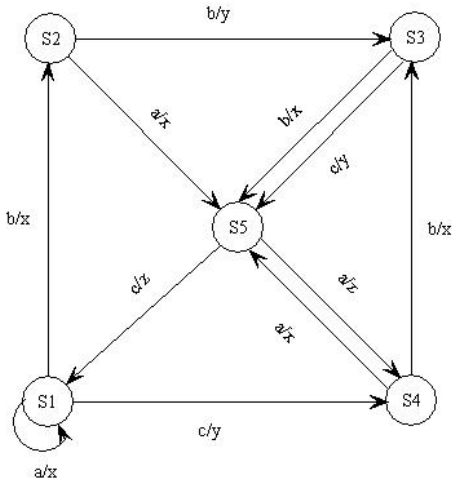


**Figure 1. A Finite State Machine**

An FSM $M$ can be viewed as a directed graph $G = (V,E)$, where the set of vertices $V$ represents the state set $S$ of $M$ and the set of edges $E$ represents the transitions. An edge has label $i/o$ where $i \in I$ and $o \in O$ are the corresponding transition's input and output. Figure 1 illustrates an FSM represented by its corresponding directed graph.

Two states $s_i$ and $s_j$ are said to be *equivalent* if and only if for every input sequence $\alpha \in I^*$ the machine produces the same output sequence, $\lambda(s_i,\alpha) = \lambda(s_j,\alpha)$. Machines $M_1$ and $M_2$ are *equivalent* if and only for every state in $M_1$ there is a corresponding state in $M_2$, and vice versa. A machine is *minimal* (*reduced*) if and only if no two states are equivalent. It will be assumed that any FSM being considered is minimal since any (deterministic) FSM can be converted into an equivalent (deterministic) minimal FSM [3]. An FSM is *completely specified* if and only if for each state $s_i$ and input $a$, there is a specified next state $s_{i+1} = \delta(s_i,a)$, and a specified output $o_i = \lambda(s_i,a)$. Otherwise, the machine is *partially specified*. An FSM is *strongly connected* if, given any ordered pair of states $(s_i,s_j)$, there is a sequence of transition that moves the FSM from $s_i$ to $s_j$.

It will be assumed throughout this article that an FSM is deterministic, minimal, completely specified, and strongly connected.

### 2.2 Conformance Testing

Given a specification FSM $M$, for which we have its complete transition diagram, and an implementation $M'$, for which we can only observe its I/O behaviour ("black box"), we want to test to determine whether the I/O behaviour of $M'$ conforms to that of $M$. This is called *conformance testing*. A test sequence that solves this problem is called a *checking sequence*. I/O behaviorial difference between specification and implementation can be caused by either an incorrect output (an output fault) or an earlier incorrect state transfer (a state transfer fault). The latter can be detected by adding final state check after a transition testing is finished. A standard test strategy is:

1. Homing: Move $M'$ to an initial state $s$;
2. Output Check: Apply an input sequence $\alpha$ and compare the output sequences generated by $M$ and $M'$ separately;
3. Tail State Verification: Using state verification techniques to check the final state.

The first step is known as homing a machine to a desired initial state. The second step checks whether $M'$ produces the desired output sequence. The last step checks whether $M'$ is in the expected state $s' = \delta(s,\alpha)$. Three techniques can be used for state verification:

- Distinguishing Sequence (DS)
- Unique Input/Output (UIO)

- Characterizing Set (CS)

A distinguishing sequence is a sequence that produces a different output for each state. Not every FSM has a DS.

A UIO sequence of state $s_i$ is an input/output sequence $x/y$, that may be observed from $s_i$, such that the output sequence produced by the machine in response to $x$ from any other state is different from $y$, i.e. $\lambda(s_i, x) = y$ and $\lambda(s_i, x) \neq \lambda(s_j, x)$ for any $i \neq j$. A DS defines a UIO. While not every FSM has a UIO for each state, some FSMs without a DS have a UIO for each state.

A characterizing set $W$ is a set of input sequences with the property that, for every pair of state ($s_i$, $s_j$), $i \neq j$, there is some $w_i \in W$ such that $\lambda(s_i, w_i) \neq \lambda(s_j, w_i)$. Thus, the output sequences produced by executing each $w_i \in W$ from $s_j$ verifies $s_j$. This paper will focus on the problem of generating UIOs.

### 2.3 State Splitting Tree

A state splitting tree is a rooted tree $T$ that is used to construct adaptive distinguishing sequences or UIOs from an FSM. Each node in the tree has a predecessor (parent) and successors (children). The predecessor of the root node, which contains the set of all states, is null. The nodes containing discrete state have empty successor. These node are also known as terminals. A child node is connected to its parent node through an edge labelled with characters. The edge implies that the states in the child node are partitioned from states in the parent node upon receiving the labelled characters. The splitting tree is complete if the partition is a discrete partition.

An example is illustrated in Figure 2 where an FSM (different from the one shown in Figure 1) has six states, namely, $S = \{s_1, s_2, s_3, s_4, s_5, s_6\}$. The input set is $I = \{a, b\}$ while output set $O = \{x, y\}$. The root node is indicated by $N(0,0)$[1], containing the set of all states. Suppose states $\{s_1, s_3, s_5\}$ produce $x$ simultaneously and arrive at some states when responding to $a$, while $\{s_2, s_4, s_6\}$ produce $y$. Then $\{s_1, s_3, s_5\}$ and $\{s_2, s_4, s_6\}$ are distinguished by $a$. Two new nodes rooted from $N(0,0)$ are then generated, indicated by $N(1,1)$ and $N(1,2)$. Continuing to input FSM with $b$, state initially from $\{s_1\}$ produces $x$ while states initially from $\{s_3, s_5\}$ produce $y$. Then $ab$ distinguish $\{s_1\}$ from $\{s_3, s_5\}$. Two new nodes rooted from $N(1,1)$ are generated, denoted $N(2,1)$ and $N(2,2)$. The same operation can be applied to $\{s_2, s_4, s_6\}$. Repeating this process, we can get all discrete partitions as shown in Figure 2 (if there is no adaptive distinguishing sequence, this will not happen). A path from a discrete partition node to the root node forms a UIO to the state related to this node. When the splitting tree is complete, we can construct UIOs for each state.

[1] $N(i,j)$: $i$ indicates that the node is in the $i^{th}$ layer from the tree. $j$ refers to the $j^{th}$ node in the $i^{th}$ layer

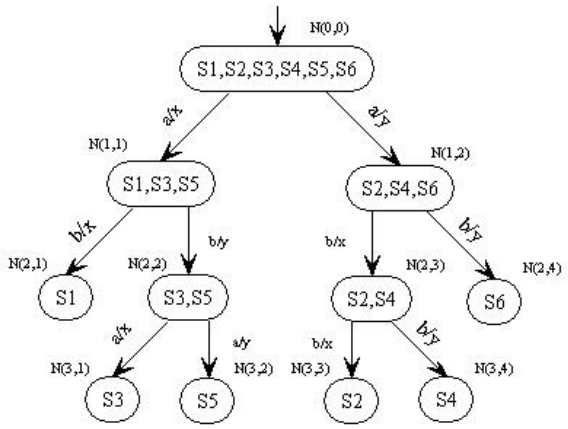


**Figure 2. A state splitting tree from an FSM**

Unfortunately, the problem of finding data to build up the state splitting tree is NP-hard. This provides the motivation for investigating the use of GAs. In the following sections, we will discuss the problem in detail.

## 3 Apply GA to FSM

### 3.1 Genetic Algorithms

A genetic algorithm (GA) is a heuristic optimisation technique that simulates natural processes, utilizing selection, crossover, mutation and fitness proportionate reproduction operators. Since Holland's seminal work (1975) [8], it has been applied to a variety of learning and optimisation problems.

A GA starts with a randomly generated population, each element (chromosomes) being a sequence of variables/parameters. Variable values can be represented in binary form, real-number,

or even characters. The quality of each chromosome is determined by a fitness function that depends upon the problem considered. Those of high fitness have a greater probability of multiple reproduction while those of low fitness have a greater probability of being rejected.

Crossover and mutation are applied to produce new chromosomes. Crossover exchanges information between randomly selected parent chromosomes by exchanging parameter values to form children. Single-point crossover, multi-point crossover and uniform crossover are three major types. Mutation injects information into the genetic pool by mutating randomly selected parameters according to the preset mutation probability. Mutation prevents genetic pool from premature convergence, namely, getting stuck in local maxima/minima. A flow chart for a simple GA is presented in Figure 3.

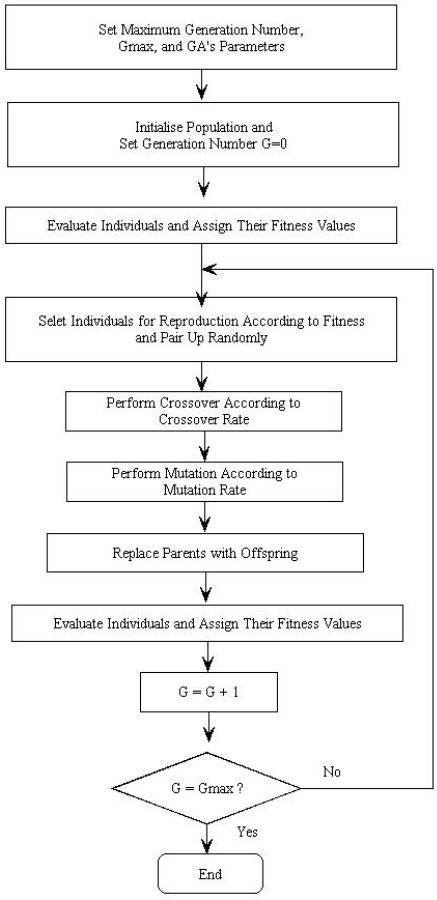


**Figure 3. Flow Chart for a Simple GA**

### 3.2 Solution Representation

When applying a GA the first question to considered is what representation should be used. In this work, the chromosomes in the genetic pool are strings of characters from the input set $I$. To preserve more information, a *DO NOT CARE* character $'\sharp'$ is also considered. We will explain the reason for using this character in the following sections. When receiving this input, the state of an FSM remains unchanged and no output is produced. Crossover operated on two parent chromosomes swaps characters. When a gene (character) is mutated according to the mutation rate, it is replaced with a character randomly selected from the rest in the input set, including $'\sharp'$.

### 3.3 Fitness Definition

A key issue is to define a fitness function to (efficiently) evaluate the quality of solutions. This function should embody two aspects: (1) Solutions should create as many discrete units as possible. (2) The solution should be as short as possible. The function needs to make a trade-off between these two points. This work uses a function that rewards the early occurrence of discrete partitions and punishes the chromosome's length. An alternative would be to model the number of state partitions and the length of solution as two objectives and then treat them as multi-object optimisation problems.

We define a fitness function in (2) that is derived from (1). While applying an input sequence to an FSM, at each stage of a single input, the state splitting tree constructed is evaluated by equation (1),

$$f_{(i)} = \alpha \frac{x_i e^{(x_i + \delta x_i)}}{l_i^{\gamma}} + \beta \frac{(y_i + \delta y_i)}{l_i}. \qquad (1)$$

where $i$ refers to the $i^{th}$ input character. $x_i$ denotes the number of existing discrete partitions while $\delta x_i$ is the number of new discrete partitions caused by the $i^{th}$ input. $y_i$ is the number of existing separated groups while $\delta y_i$ is the number of new groups. $l_i$ is the length of the input sequence up to the ith element (***Do Not Care*** characters are excluded). $\alpha$, $\beta$ and $\gamma$ are constants. From the equation it can be seen that, when $x_i$ increases, $f_{(i)}$ will exponentially increase, while, when an input sequence's length $l_i$ increases, $f_{(i)}$ is reduced exponentially. Suppose $x_i$ and $l_i$ change

approximately at the same rate, that is $\delta x_i \approx \delta l_i$, as long as $e^{(x_i+\delta x_i)}$ has faster dynamics than $l_i^\gamma$, $\frac{x_i e^{(x_i+\delta x_i)}}{l_i^\gamma}$ will still increase exponentially. If no discrete partition is found with the input sequence length increases, the fitness function will decrease dramatically. $\frac{x_i e^{(x_i+\delta x_i)}}{l_i^\gamma}$ thus performs two actions: encouraging the early occurrence of discrete partitions and punishing the increment of an input sequence's length. $\frac{(y_i+\delta y_i)}{l_i}$ will also affect $f_{(i)}$ in a linear way. Comparing to $\frac{x_i e^{(x_i+\delta x_i)}}{l_i^\gamma}$, it plays a less important role. This term rewards partitioning even when discrete classes have not been produced.

After all input characters have been examined, the final fitness value for this input candidate is defined as the average of (1)

$$F = \frac{1}{N}\sum_{i=1}^{N} f_{(i)}. \qquad (2)$$

where $N$ is the sequence's length.

### 3.4 Tracking Historical Records

Mutation prevents a GA from getting stuck in a local maxima/minima but might also force a GA to jump out of the global maxima/minima when it happens to be there. Solutions provided at the end of evolutionary computation could be good, but need not be the best found during the process. It is therefore useful to keep track of those candidates that have produced good or partially good solutions, and store them for the purpose to further optimise the final solutions.

Consider an example shown in Figure 4. Suppose that a GA produces a UIO sequence $U_t$ for state $s_t$, forming a path shown in thin solid arrow lines. During the computation, another solution $U_t'$ for $s_t$ has been found, forming a path shown in dotted arrows. The two lines visit a common node at $N_4$. $U_t$ has a shorter path than $U_t'$ before $N_4$ while has a longer path after $N_4$. The solution recombined from $U_t$ and $U_t'$ (indicated in figure by thick arrow lines), taking their shorter parts, is better than either of them.

In this work, a database is created to track all candidates that result in the occurrence of discrete partitions. This database is then used to further optimise the final solutions through recombination. Solutions for a state, which are of the same length, are multi-UIOs of this state which can be used in test generation [9, 10].

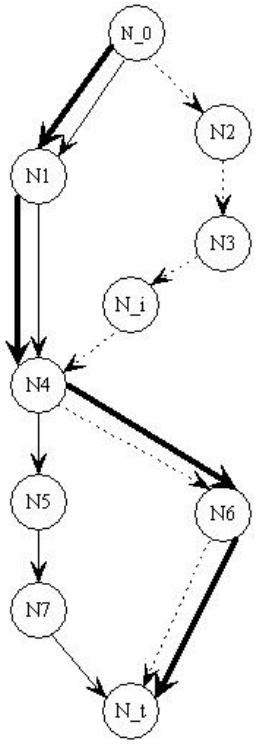


**Figure 4. Solution Recombination**

## 4 Experiments

A set of experiments was devised to test the GA's performance. The first FSM studied is the one shown in Figure 1. All minimum-length UIOs of all states are presented in Table 1. Roulette Wheel Selection (RWS) and Uniform Crossover (UC) are implemented.

| State | UIOs |
|---|---|
| $s_1$ | aa/xx, ab/xx, ac/xy, ba/xx bb/xy, ca/yx, cb/yx |
| $s_2$ | b/y |
| $s_3$ | ba/xz, bc/xz, ca/yz, cc/yz |
| $s_4$ | bb/xx, bc/xy |
| $s_5$ | a/z, c/z |

**Table 1.** UIO Sequences for Figure 1

In the first experiment, the input space is $I = \{a, b, c\}$. The parameters are set to: [2] $ChrLen = 10$, $XRate = 0.75$, $MRate = 0.05$, $PSize = 30$, $MGen = 50$, $\alpha = 0.1$, $\beta = 1.5$, and $\gamma = 5$.

[2] ChrLen:Chromosome Length; XRate:Crossover Rate; MRate:Mutate Rate; PSize: Population Size; MGen:Max Generation

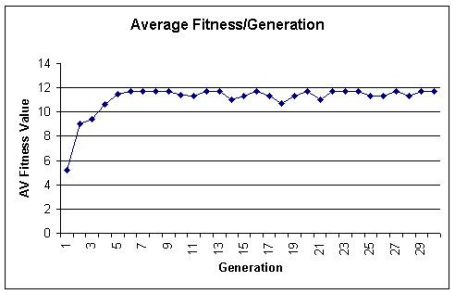


**Figure 5.** Input Space {a,b,c}

At the end of computation, by looking at the average fitness values (Figure 5), we found that the genetic pool converges quite quick. The curve is comparatively smooth. However, by examining all individuals (Table 2), we found that the whole population tends to move to the individuals that start with *bb* or *bc*. The population loses its diversity and converges prematurely. Consequently, only a few UIOs have been found {*b*, *bb*, *bc*}.

| ID | Sequence | ID | Sequence |
|---|---|---|---|
| 1 | *bccaabcacc* | 16 | *bbcaabcacc* |
| 2 | *bbbaabaacc* | 17 | *bbbaabaaca* |
| 3 | *bbbaabcacc* | 18 | *bbcaababcc* |
| 4 | *bcccabaacb* | 19 | *bbbcabbacc* |
| 5 | *bbcbabcacb* | 20 | *bbcaabaacc* |
| 6 | *bbbaabcacc* | 21 | *bccaabaacc* |
| 7 | *bbcbabcacb* | 22 | *bbbaabaacc* |
| 8 | *bbccabcacb* | 23 | *bbbcabbacc* |
| 9 | *bbcbabaacb* | 24 | *bbcbabcacb* |
| 10 | *bcbbabaacb* | 25 | *bbbcabcacc* |
| 11 | *bbcbabcabb* | 26 | *bbcaabcacb* |
| 12 | *bbcaabaacc* | 27 | *bbbbababcc* |
| 13 | *bccaabaacb* | 28 | *bbcaabcacb* |
| 14 | *bbbbabcacc* | 29 | *bbccabbacc* |
| 15 | *bbcbabcacb* | 30 | *bbbaabbacb* |

**Table 2.** Final sequences from input space {a,b,c}

This is not what we expected. To keep the genetic pool diverse, we introduced a ***DO NOT CARE*** character $'\sharp'$. When receiving this character, the state of an FSM remains unchanged. The input space is then $\{a, b, c, \sharp\}$. We keep the same values for all other parameters. The average fitness chart is presented in Figure 6. It can be seen that Figure 6 is not as smooth as Figure 5, but still shows a general tendency to increase. After examining the genetic pool, we found that eleven UIOs, $\{a, b, c, ab, ac, ba, bb, bc, ca, cb, cc\}$, were found ([3]Table 3). By retrieving the historical records, we also found $\{aa\}$ (Table 4). The GA thus performs well in this experiment.

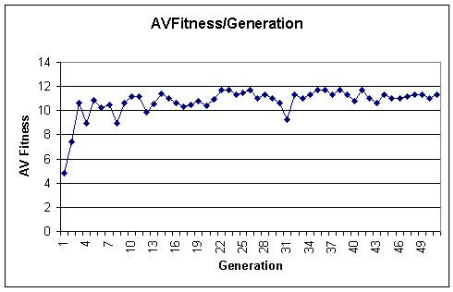


**Figure 6.** Input Space {a,b,c,♯}

| ID | Sequence | VS | ID | Sequence | VS |
|---|---|---|---|---|---|
| 1 | *bbbbcccb♯c* | *bb* | 16 | *♯bcbabcabb* | *bc* |
| 2 | *ab♯bbc♯bbc* | *ab* | 17 | *♯♯cba♯cbcb* | *cb* |
| 3 | *♯bc♯caaaba* | *bc* | 18 | *♯bb♯♯bbaca* | *bb* |
| 4 | *♯♯bcacbc♯c* | *bc* | 19 | *bc♯bb♯cbb♯* | *bc* |
| 5 | *b♯bcaca♯cb* | *bb* | 20 | *cbcbbc♯♯ca* | *cb* |
| 6 | *bb♯bca♯♯bb* | *bb* | 21 | *♯bc♯aabc♯c* | *bc* |
| 7 | *cbacc♯ac♯b* | *cb* | 22 | *c♯a♯cacc♯c* | *ca* |
| 8 | *acabaaaacc* | *ac* | 23 | *c♯accac♯♯a* | *ca* |
| 9 | *ba♯a♯aaa♯b* | *ba* | 24 | *♯b♯bcabbc♯* | *bb* |
| 10 | *♯ccaab♯acc* | *cc* | 25 | *bb♯bba♯♯a♯* | *bb* |
| 11 | *♯cbc♯aa♯ab* | *cb* | 26 | *bacb♯c♯b♯b* | *ba* |
| 12 | *♯bbcca♯aaa* | *bb* | 27 | *bb♯bb♯♯♯c♯* | *bb* |
| 13 | *cccb♯♯♯♯♯c* | *cc* | 28 | *cb♯babc♯b♯* | *cb* |
| 14 | *aca♯bbaa♯b* | *ac* | 29 | *cccc♯cc♯bb* | *cc* |
| 15 | *cbb♯c♯cb♯c* | *cb* | 30 | *b♯cca♯♯b♯c* | *bc* |

**Table 3.** Final sequences from input space {a,b,c,♯}

| ID | Sequence | VS | Fitness |
|---|---|---|---|
| 1 | *aa♯caab♯c* | *aa* | 6.2784 |
| 2 | *a♯aba♯bc♯c* | *aa* | 4.2605 |

**Table 4.** Historical Records

The reason that crossover operator can be used to explore information is that it forces genes to move among chromosomes. Through recombination of genes, unknown information can be uncovered by new chromosomes. However, the gene movement exerted by crossover operator can only happen among different chromosomes. We call it

[3] *Sequence*: candidate sequence. *VS*: minimum-length UIO.

vertical movement. By using a *DO NOT CARE* character, some spaces can be added in a chromosome, which makes it possible for genes to move horizontally. Therefore, *DO NOT CARE* makes the exploration more flexible, and, consequently, can help to keep the genetic pool diverse.

We organised eleven experiments with the same parameters. By examining the solutions obtained in the final genetic pool (historical records are excluded), we evaluated the average performance. Table 5 shows that, in the worst case, 8 out of 12 UIOs are found, which accounts for 66.7%. The best case is 100%. The average is 86.4%.

| Exp. | UIOs Found | Total | Percent(%) |
|---|---|---|---|
| 1 | 8 | 12 | 66.7 |
| 2 | 8 | 12 | 66.7 |
| 3 | 11 | 12 | 91.7 |
| 4 | 11 | 12 | 91.7 |
| 5 | 10 | 12 | 83.3 |
| 6 | 11 | 12 | 91.7 |
| 7 | 11 | 12 | 91.7 |
| 8 | 11 | 12 | 91.7 |
| 9 | 10 | 12 | 83.3 |
| 10 | 12 | 12 | 100 |
| 11 | 11 | 12 | 91.7 |
| Avg | 10.36 | 12 | 86.4 |

**Table 5.** Different Experiment Results

After examining the solutions from different experiments, we found that *aa* is the hardest UIO to be found while *bb* and *bc* are most frequent ones that occur in the final solutions. By checking Table 1., we found a very interesting fact: a majority of UIOs initially start with *b* or *c*. If individuals happen to be initialised with $ba \times \times \times \times \times \times \times \times$, they will distinguish $s_1$, $s_2$ and $s_5$ in the first two steps, and so achieve high fitness. These individuals are likely to be selected for the next generation. Individuals initialised with $aa \times \times \times \times \times \times \times \times$ can distinguish only $s_1$ and $s_5$ in the first two steps, and achieve lower fitness values. They are less likely to be selected for reproduction. This fact seems to imply that there exist multiple modals in the search space. Most individuals are likely to crowd on the highest peak. Only a very few individuals switch to the lower modals. To overcome this problem, *sharing techniques* might help. The application of such approaches will be left for future work.

We then turn to compare the performance between GA and random search (RS). RS is defined as randomly perturbing one bit in an input sequence. 30 input sequences of ten input characters were randomly generated. We repeated this experiment several times. The results shown in Table 6 are the best ones. From the table it can be seen that 11 out of 12 UIOs (a, b, c, aa, ab, ac, bb, bc, ca, cb, cc) are found over these experiments. Only one is missed (*ba*).

| ID | Sequence | VS | ID | Sequence | VS |
|---|---|---|---|---|---|
| 1 | *cabbbcbcac* | *ca* | 16 | *cacacacccc* | *ca* |
| 2 | *ccbcbbbcbc* | *cc* | 17 | *bcbbabcbca* | *bc* |
| 3 | *bcccaaabcb* | *bc* | 18 | *ccccbbcccb* | *cc* |
| 4 | *cccccbcbac* | *cc* | 19 | *accbbccacb* | *ac* |
| 5 | *ccbbacbccc* | *cc* | 20 | *cbccaccccb* | *cb* |
| 6 | *bccccabcac* | *bc* | 21 | *acbbcbbbbb* | *ac* |
| 7 | *bccbcbbcbc* | *bc* | 22 | *cbccccbccb* | *cb* |
| 8 | *ccbbccaabc* | *cc* | 23 | *ccabacbcca* | *cc* |
| 9 | *bbacccccba* | *bb* | 24 | *accacbabcc* | *ac* |
| 10 | *cbbbbacccb* | *cb* | 25 | *aabcbacbbb* | *aa* |
| 11 | *bcaacccccb* | *bc* | 26 | *cabbacacbc* | *ca* |
| 12 | *cccbcbbcbc* | *cc* | 27 | *bcbccccaac* | *bc* |
| 13 | *abcbbcccab* | *ab* | 28 | *aabcbccbca* | *aa* |
| 14 | *bccccbccc* | *bc* | 29 | *bccccaabbb* | *bc* |
| 15 | *bbccbccbba* | *bb* | 30 | *acaccbaaba* | *ac* |

**Table 6.** Solutions by random search

Since the FSM is comparatively simple, and the UIOs are short, it is not difficult to find all UIOs through RS. Thus the GA does not show significant advantages over RS. A more complicated system, shown in Figure 7, is therefore designed to further test GA's performance. Unfortunately, no existing UIOs are available, which means that we can never be sure that a complete set of UIOs has been found. Hence, we will compare the numbers of UIOs found by using RS and the GA separately.

A total of 50 candidates were used in the experiment. All UIOs found, whether minimum-length or not, are listed to make a comparison. Experiments on both RS and the GA were repeated several times. The solutions presented in Table 7 and Table 8 are the best ones. Table 7 lists the UIOs obtained through RS while Table 8 shows the solutions found by GA. After comparing these two tables, we find that the GA finds many more UIOs than RS does. Both RS and GA easily find the short UIOs. However, for other UIOs the performance of GA appears to be much better than that of RS. For example, the GA finds

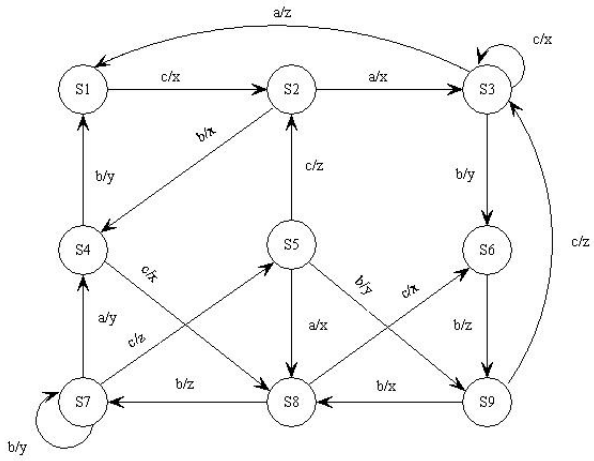


**Figure 7. A more complicated FSM**

*bcbcc*/*xxzzz* *cbcbb*/*xzzyz* while RS does not.

We also measure the frequency of hitting UIOs. RS is redefined by initialising population routinely. Experimental result show that both methods hit UIOs with the length of 3 or less frequently. However, on hitting those with the length of 4, RS is roughly the half times of GA, while, for those with the length of 5, in the first 30 iterations, RS hits 10 times while GA 27. All these results suggest that, in simple systems, it is possible to obtain good solutions through random search. However, in more complicated systems, especially in those with large input and state spaces, finding UIOs with random search is likely to be infeasible. By contrast, GA seems to be more flexible. In future work, we will apply GA to some examples with high input dimensions, or some real applications such as the *IEEE 802.2 Logical Level Control (LLC) Protocol* (48 inputs & 65 outputs).

| State | UIOs |
|---|---|
| $s_1$ | ca/xx, cb/xx |
| $s_2$ | aa/xz, ab/xy, acc/xxx, bb/xy bcc/xxx, bcba/xxzy |
| $s_3$ | a/z, bb/yz, ca/xz, cb/xy, ccb/xxy, ccc/xxx |
| $s_4$ | bc/yx, cba/xzy, ccb/xxz cbca/xzzx, cbcc/xzzz |
| $s_5$ | ab/xz, acb/xxz, bb/yx bcc/yzx, cacc/zxxx, cb/zx |
| $s_6$ | bb/zx, bcc/zzx |
| $s_7$ | a/y, bb/yy, bcc/yzz, bcbc/yzyz, cbc/zyz, cc/zz |
| $s_8$ | bb/zy, bcc/zzz, bcbc/zzyz, cbca/xzzz, cbcc/xzzx |
| $s_9$ | bb/xz, ca/zz, cc/zx |

**Table 7.** UIO Sequences By Random Search

| State | UIOs |
|---|---|
| $s_1$ | ca/xx, cb/xx |
| $s_2$ | aa/xz, ab/xy, aca/xxz,acb/xxy acc/xxx, bb/xy, bcc/xxx, bcbcc/xxzzz |
| $s_3$ | a/z, bb/yz, ca/xz, cb/xy, ccb/xxy, ccc/xxx |
| $s_4$ | bc/yx, cba/xzy,cbcbb/xzzyx cbb/xzy, ccb/xxz, cbca/xzzx, cbcc/xzzz, cbcbc/xzzyz |
| $s_5$ | ab/xz, acb/xxz, bb/yx, bca/yzz bcc/yzx, cab/zxy, cb/zx |
| $s_6$ | bb/zx, bca/zzz, bcc/zzx |
| $s_7$ | a/y, ba/yy, bb/yy, bca/yzx bcc/yzz, cab/zxz, cbb/zyx, cc/zz cbc/zyz, cc/zz, bcbc/yzyz |
| $s_8$ | ba/zy, bb/zy, bca/zzx, bcc/zzz, cbb/xzx, cbca/xzzz, cbcc/xzzx, cbcbb/xzzyz |
| $s_9$ | bb/xz, ca/zz, cbb/zyz, cc/zx |

**Table 8.** UIO Sequences Found by GA

## 5 Conclusion

In this paper, we investigated GA's performance in computing UIOs from an FSM. We showed that the fitness function can guide the candidates to explore potential UIOs by encouraging the early occurrence of discrete partitions while punishing length. We showed that using *DO NOT CARE* character can help to improve the diversity in GAs. Consequently, more UIOs can be explored. The simulation results in a small system showed that, in the worst case, 67% of the minimum-length UIOs have been found while, in the best case, 100%. On the average, more than 85% minimum-length UIOs were found from the model under the test. In a more complicated system, GA found many more UIOs than random search. The GAs was much better, than random search, at finding the longer UIOs. These experiments and figures suggest that GAs can provide good solutions on computing UIOs.

We also found that some UIOs were missed with high probability. This may be caused by their lower probability distribution in the search space. Future work will consider this problem.

## References


[1] A.V.Aho, A.T.Dahbura, D.Lee and M.U.Uyar, "An Optimization Technique for Protocol Conformance Test Generation Based on UIO Sequences and Rural Chinese Postman Tours" *IEEE Trans, Communications*, vol.39, No.3, pp.1604-1615, 1991.

[2] D.Lee and M.Yannakakis, "Testing Finite State Machines: State Identification and Verification" *IEEE Trans, Computers*, vol.43, No.3, pp.306-320, 1994.

[3] D.Lee and M.Yannakakis, "Principles and Methods of Testing Finite State Machines - A Survey" *Proceedings of IEEE*, vol.84, No.8, pp.1090-1122, 1996.

[4] D.P.Sidhu, and T.K.Leung, "Formal Methods for Protocol Testing: A Detailed Study" *IEEE Transactions on Software Engineering*, Vol.15, No.4, April 1989.

[5] Kshirasagar Naik, "Efficient Computation of Unique Input/Output Sequences in Finite-State Machines" *IEEE/ACM Transactions on Networking*, vol.5, No.4, pp.585-599, Aug., 1997.

[6] Z.Kohavi, "Switching and Finite Automata Theory", *New York: McGraw-Hill*, 1978.

[7] D.E.Goldberg, "Genetic Algorithms in Search, Optimization, and Machine Learning". Reading, MA: Addison-Wesley, 1989.

[8] J.H.Holland, "Adaptation in Natural and Artificial Systems". Ann Arbor, MI, University of Michigan Press, 1975.

[9] Y.N.Shen, F.Lombardi, and A.T.Dahbura, "Protocol Conformance Testing Using Multiple UIO Sequences" *Proc, Ninth IFIP WG6.1 Int. Symp. on Protocol Specif., Test., and Verif.*, 1989.

[10] Bo Yang and Hasan Ural, "Protocol Conformance Test Generation Using Multiple UIO Sequences with Overlapping" *ACM SIGCOMM 90: Communications, Architectures, and Protocols*, Twente, The Netherlands, Sep.24-27, pp118-125. North-Holland, The Netherlands.

[11] R.E.Miller and S.Paul, "On the Generation of Minimal-Length Conformance Tests for Communication Protocols" *IEEE/ACM Transactions on Networking*, Vol.1, No.1, Feb, 1993.

[12] Xiaojun Shen and Guogang Li, "A new protocol conformance test generation method and experimental results" *Proceedings of the 1992 ACM/SIGAPP Symposium on Applied computing*,pp.75-84, Kansas City, Missouri, United States, 1992.