Advance Access published on March 10, 2006 doi:10.1093/comjnl/bxl003

Automated Unique Input Output Sequence Generation for Conformance Testing of FSMs

KARNIG DERDERIAN[1,*], ROBERT M. HIERONS[1], MARK HARMAN[2] AND QIANG GUO[3]

[1]*Department of Information Systems and Computing, Brunel University, Uxbridge, Middlesex UB8 3PH, UK*
[2]*Department of Computer Science, King's College London, London WC2R 2LS, UK*
[3]*Department of Computer Science, University of Sheffield, Sheffield S1 4DP, UK*
[*]*Corresponding author: karnig.derderian@brunel.ac.uk*

This paper describes a method for automatically generating unique input output (UIO) sequences for FSM conformance testing. UIOs are used in conformance testing to verify the end state of a transition sequence. UIO sequence generation is represented as a search problem and genetic algorithms are used to search this space. Empirical evidence indicates that the proposed method yields considerably better (up to 62% better) results compared with random UIO sequence generation.

Keywords: Finite state machine, unique input output sequence, state verification, conformance testing, genetic algorithm

Received 22 December 2004; revised 30 January 2006

1. INTRODUCTION

Finite state machines (FSMs) have been used to model systems in different areas like sequential circuits [1], software development [2] and communication protocols [3, 4, 5, 6, 7, 8, 9, 10]. To ensure the reliability of these systems once implemented they must be tested for conformance to their specification. Usually the implementation of a system specified by an FSM is tested for conformance by applying a sequence of inputs and verifying that the corresponding sequence of outputs is that which is expected.

Ideally complete test suites are produced that would distinguish any faulty implementation given that it does not have more states than its specification. However, often this is not feasible because these methods rely on FSM with certain characteristics that cannot always be guaranteed. Work on generating complete test suits relies on either a distinguishing sequence (DS) being present in an FSM [1, 11, 12, 13], the existence of a reliable reset in the FSM [2] or generation of test sequences of at least exponential (in terms of the number of states) length [14]. These issues will be later discussed in the paper. Hence generating incomplete test suites has been of interest.

This paper focuses on the U-method for test sequence generation [15] where unique input/output (UIO) sequences for each state have to be generated. The problem of generating such sequences is known to be NP-hard [16]. While a random algorithm could be used it does not always produce acceptable results. Representing test sequence generation as a search problem with a specified fitness function gives the opportunity for algorithms known to be robust in searches of unknown domains, such as genetic algorithms (GAs) [17], to be used. Generating test sequences using such algorithms could provide a computationally easy solution that produces good results as shown by [18].

One of the primary contributions of this paper is the proposal for a more computationally efficient and yet effective method of generating UIO sequences. The proposed method also does not suffer from the usual restriction of some test sequence generation methods (D-method and W-method for example) where only fully specified FSMs can be considered. The generated UIOs can be used for partially or completely specified FSMs that in turn can be used in generating a test sequence using the U-method. As a result weak conformance testing can be applied to partially specified FSMs without having any completeness assumption.

In order to minimize manual testing and hence software production costs and speed the process up, automation is necessary. Automation has been widely used in testing and test data generation [6, 15, 19, 20]. Automating the generation of UIO sequences can contribute to this.

The primary contributions of this paper show how UIO generation can be formulated in terms of an automated search problem and describe an approach to automation of UIO generation using GAs. This paper demonstrates that UIO generation can be reduced to an automated search problem and presents results from an empirical study of this approach.

The paper presents results from an empirical study of this approach, which provides evidence that the GA is successful in guiding the automated search.

The paper begins with some preliminaries on FSMs, conformance testing and GAs in Section 2. Section 3 shows how the UIOs can be generated using GAs and the results from the conducted experiments are presented in Section 4. Finally in Section 5 conclusions are drawn.

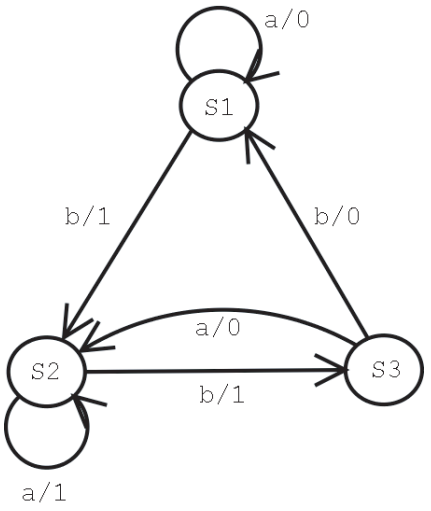

FIGURE. 1. Transition diagram of an FSM $M1$.

2. BACKGROUND

Testing is an important part of the software engineering process and can account for up to 50% of the total cost of software development [21]. This motivates the study of testing FSMs to ensure the correct functioning of systems.

The generation of efficient and effective test sequences is very important in conformance testing. Test sequences can be generated using formal methods like transition tours (T-method), UIO sequences (U-method), DSs (D-method) and characterizing sets (CSs) (W-method). The U-methods are popular because of the following reasons [5]. The T-method does not consider state transfer faults since it does not verify the final state of a transition sequence. The W-method relies on a reliable reset for the FSM and in practice UIOs lead to shorter test sequences than those produced using CSs. There exist FSMs with UIOs for every state but no DS. Practitioners report that in practice many FSMs have UIOs [3].

2.1. Finite state machines

Finite state systems are usually modelled using Mealy machines that produce an output for every transition triggered by an input. An FSM M can be denoted $M = (S, s_1, \delta, \lambda, X, Y)$ where S, X, Y are finite non-empty sets of states, input symbols and output symbols respectively and $s_1 \in S$ is the initial state. δ is the state transition function and λ is the output function. A transition is represented as $t = (s_i, x, y, s_j)$ where $s_i \in S$ is the start state, $s_j \in S$ is the end state, $x \in X$ is the input and $y \in Y$ is the output. When a machine M in state $s_i \in S$ receives input x it moves to state $\delta(s_i, x) = s_j$ and outputs $\lambda(s_i, x) = y$. The functions δ and λ can be extended to take input sequences to give functions δ^* and λ^* respectively. FSMs can be represented using state transition diagrams where the vertices correspond to states and the edges to state transitions which are labelled with the associated input and output [16] (Figure 1).

An FSM is said to be deterministic if there is no pair of transitions that have the same initial state and input i.e. upon an input a unique transition follows to the next state. If for any state an input could trigger more than one transition the machine is non-deterministic. FSMs for which a transition exists for every input $a \in X$ and state $s \in S$ are known as completely (fully) specified. Given an FSM that is partially specified it is possible to take a completeness assumption and complete M by either adding an error state or assuming that where the input was not specified originally an empty output should be produced.

Those FSMs where every state can be reached from the initial state are known as initially connected. Unreachable states can be removed from any FSM to make it initially connected. An FSM M is strongly connected if for every pair of states (s_i, s_j) from M there is some input sequence that takes M from s_i to s_j. If M is initially connected and has a reset operator then it must be strongly connected. A reset operator takes the FSM to its initial state. The presence of a correctly implemented reset operator is sometimes important for transition testing but cannot always be guaranteed. One of the advantages of the U-method is that it does not need a reset.

Two states s_i and s_j are said to be equivalent if for every input sequence the same output sequence is generated. Otherwise the two states are inequivalent and there exists an input sequence x where $\lambda^*(s_i, x) \neq \lambda^*(s_j, x)$ and that sequence is known as a separating sequence. Comparing states from different machines is similar. Two FSMs M and M' are equivalent if their initial states are equivalent. A minimal FSM

is a machine M such that there is no equivalent FSM M' with fewer states than M.

For example Figure 1 represents the deterministic FSM $M1$. $M1$ with an initial state s_1 is initially and strongly connected as every state in $M1$ is reachable from any other state. $M1$ is also completely specified and minimal.

In this work we consider only deterministic FSMs. For non-deterministic FSM conformance testing refer to [22, 23, 24]. It is also safe to assume that only minimal FSMs should be considered as any deterministic FSM can be minimized [25] and there are well-known methods to automatically do so [25, 26]. Also for the reasons outlined before only strongly connected FSMs are considered.

2.2. Conformance testing

When testing from an FSM model M it is assumed that the implementation under test (IUT) can be modelled by an unknown FSM M' and thus that testing involves comparing the behaviour of two FSMs. Verifying that M' is equivalent to M by only observing the input/output behaviour of M' is known as conformance testing or fault detection.

Often a fault can be categorized as either an output fault or a state transfer fault. Output faults are those faults where the wrong output is produced and state transfer faults are those faults where the state after a transition is wrong. An output fault can be detected by executing a transition and observing its output. A state transfer fault can be detected by checking if the final state is correct after the transition testing is applied. Suppose we wish to check a transition $t = (s_i, x, y, s_j)$. The test strategy would involve moving M' to s_i, applying the input x, verifying that the output is y, and using a state verification technique to verify the transition's end state [2].

The first step is known as homing a machine to a desired initial state s_i. It can be done by using a homing sequence which can be constructed in polynomial time [26]. The second step, transition verification, is to check whether M' produces a desired output sequence. The last step is to check whether M' is in the expected state $s_j = \delta(s_i, x)$. There are three main techniques that can be used in state verification:

- DS
- UIO sequence
- CS

A DS is an input sequence that produces output unique for each state. Not all FSMs have a DS.

A UIO for state s is an input/output sequence x/y such that $\lambda^*(s, x) = y$ and $\forall s' \in S.s' \neq s$, we have that $\lambda^*(s', x) \neq y$. A DS defines a UIO for every state. While not every FSM has UIOs for all states, some FSMs without a DS have UIOs for all states. Also in practice most FSMs have UIOs for all states [7].

A CS is a set of input sequences W which can distinguish any pair of states. If every sequence in W is executed from some state s_j, the set of output sequences verifies s_j. However this technique requires a number of input sequences to be executed for each state, and therefore could lead to long test sequences. For some states not every element of W is required and some subset can be used (the Wp method). This can reduce the effort involved in verifying a state. Some improvements on the W-method are presented in [22, 27, 28].

A general method for constructing minimal length checking sequences described in [29] utilizes DSs, CSs or UIOs depending on their existence.

In order to minimize test sequence length when testing using UIOs, usually minimal UIOs are used (the shortest UIO for a state). However it has been suggested [30] that using non-minimal UIOs can improve the chance of avoiding fault masking (when two or more faults collectivelý mask their faulty behaviour leading to false confidence in the IUT). Different UIOs for the same state can be compared by using a metric known as degree of difference (DoD) [31]. The DoD between two transition walks with identical input sequence is defined as the number of output differences between them. A UIO with higher DoD is expected to be more fault tolerant [30].

Some UIOs could be of exponential length. Generally if a UIO is longer than of $O(n^2)$ it might not be worth considering since a CS with upper bound of $O(n^2)$ length would exist [2].

Not all FSMs are completely specified. There are two types of conformance testing, strong and weak, depending on how unspecified transitions are treated. In strong conformance testing a completeness assumption stating how missing transitions are to be treated is necessary for partially specified FSMs. In weak conformance testing the missing transitions are treated as 'don't care' and the implementation is required to have only the same 'core behaviour' as the specification.

UIOs have been popular [5, 32] since they help in state transition fault detection and tend to yield shorter test sequences than the D and W methods [5, 32]. UIOs do not necessarily need a reliable reset operator. Only the U-method and the T-method can be used for weak conformance testing of partially specified FSMs [5]. However the T-method does not check for state transition faults.

In order to test a transition of an FSM the machine has to be put in the initial state of that transition. Then the input is applied and the output checked to verify that it is as expected. After that the UIO sequence for that state is used to verify that there is no state transfer fault. Several test sequence generation techniques based on UIOs can be used [3, 5, 6, 7, 9, 33]. This motivates an interest in automating the generation of UIOs.

2.3. Genetic algorithms

A GA [17, 34] is a heuristic optimization technique which derives its behaviour from a metaphor of the processes of

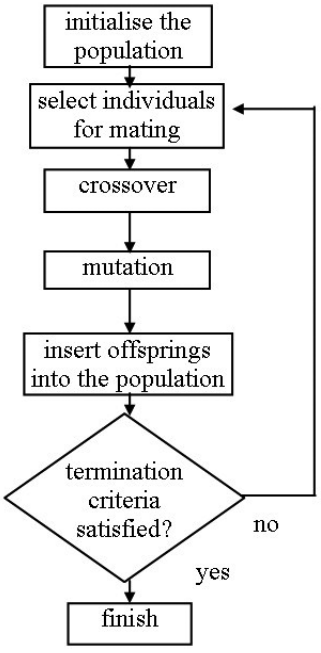

FIGURE 2. Flowchart for a basic GA.

evolution in nature. GAs have been widely used in search optimization problems [17]. GAs and other meta-heuristic algorithms have also been also used to automate software testing [19, 20, 35, 36, 37]. GAs are known to be particularly useful when searching large, multimodal and unknown search spaces. One of the benefits of GAs is their ability to escape local minima in the search for the global minimum.

Generally a GA consists of a group of individuals (population of genomes), each representing a potential solution to the problem in hand. An initial population with such individuals is usually selected at random. Then a parent selection process is used to pick a few of these individuals. New offspring individuals are produced using crossover, which keeps some of their parent's characteristics and mutation, which introduces some new genetic material. The quality of each individual is measured by a fitness function, defined for the particular search problem. Crossover exchanges information between two or more individuals. The mutation process randomly modifies offspring individuals. The population is iteratively recombined and mutated to evolve successive populations, known as generations. When the termination criterion specified is satisfied, the algorithm terminates. A flowchart for a simple GA is presented in Figure 2.

There are many different types of GAs, but they all share the basic principle of having a pool (population) of potential solutions (genomes) where some are picked using a biased selection process and recombined by crossover and mutation operations. An objective function, known as the fitness function, defines how close each individual is to being a solution and hence guides the search.

When using a GA to solve a problem the first issue that needs to be addressed is how to represent potential solutions in the GA population. A genotype is how a potential solution is encoded in a GA, while the phenotype is the real representation of that individual. There are different representation techniques, the most common being binary and characters. Gray coding is a binary representation technique that uses slightly different encoding to standard binary. It has been shown [38] that Gray codes are generally superior to standard binary by helping to represent the solutions more evenly in the search space.

The first step in a GA involves the initialization of a population of usually randomly generated individuals. The size of the population is specified at the start. Every individual is evaluated using the fitness function. When ranking is used the population is sorted according to the fitness value of the individuals. Then each individual is ranked irrespective of its size and its predecessors' fitness. This is known as linear ranking. It has been shown that using linear ranking helps reduce the chance of a few very fit individuals dominating the search leading to a premature convergence [39].

An important part of the algorithm is parent selection. A commonly used technique is the roulette-wheel selection. Here the chance of an individual being selected is directly proportional to its fitness or rank (if linear ranking is used). Hence the selection is biased towards fitter individuals.

A genome is made up of one or more chromosomes, each representing a parameter in the fitness function. In some literature genome is referred to as chromosome and genes refer to what we call chromosomes, but here we use chromosome as a part of a genome and gene as the building block of a chromosome.

The most common recombination technique used is crossover. During crossover the genes of the two parents are selectively used to create one or more new offsprings. The simplest crossover is known as single point crossover [39]. For example Figure 3 shows how a single point crossover is applied to two parent chromosomes where two new child chromosomes are produced. There is also multiple point crossovers [40]. In this work single point crossover was used with a randomly generated crossover point as used in [41].

Mutation is applied to each individual after crossover. It randomly alters one or more genes known as single point and multiple point mutation respectively [17]. Not all individuals are mutated. A predefined mutation rate (typically the reciprocal of the chromosome length) is used to determine if mutation will be performed. A single point mutation with randomly selected point was used in this work as in [41].

There can be different termination criteria for a GA depending on the fitness function. If the fitness function is such that a solution would produce a specific fitness value, which is known, then the GA can terminate when an individual

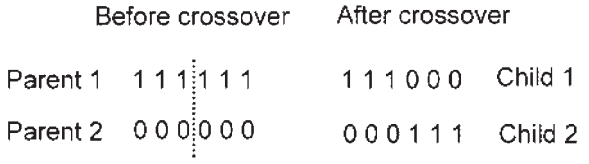

FIGURE 3. Example of crossover.

with such fitness is generated. However in many cases this is not known therefore the GA must be given other termination criteria. Such a criterion can be the specification of a maximum number of generations after which the GA will terminate irrespective of whether a solution has been generated. Another commonly used termination criterion is population saturation. After the fitness of all or some of the individuals in the GA population has not increased for a number of generations, it is assumed that a peak of the search space has been found that cannot be escaped. Usually a combination of these termination criteria is used. We use all three.

3. UIO SEQUENCE GENERATION

The problem of constructing UIO sequences is known to be NP-hard [16]. While a random search algorithm would be cheap to implement, it does not always produce acceptable results. Representing UIO sequence generation as a search problem with a specified fitness function gives the opportunity for algorithms known to be robust in searches of unknown domains, such as GAs, to be used.

A UIO for a given state s of an FSM is an input/output sequence that labels a sequence of transitions from s, but does not label a sequence of transitions from any other state. The UIOs considered in this work do not contain transitions unspecified in the FSM specification. This allows for weak conformance testing of partially specified machines. The proposed method uses GA search in an attempt to generate a UIO sequence for each state of a given FSM. A fitness function directs the search. The fitness function estimates how likely it is that a given transition sequence is a UIO sequence without actually verifying that it is one. For an input sequence of size l for a given state in an FSM with n states the fitness function used is of $O(l)$ complexity while a UIO verification algorithm would be of $O(nl)$.

Previous work [18] has shown that a GA may be used in the generation of UIOs using a state splitting tree. A state splitting tree is a rooted tree that is used to construct adaptive DSs or UIOs from an FSM. The fitness function encourages candidates to split the set of all states (in the root) into more discrete units (that share the same input and output characters). Hence the fitness function guides the search to explore potential UIOs by rewarding the early occurrence of discrete partitions while penalizing the length of the sequence. The previous work differs from that described here in three important ways: (i) it used a more computationally intensive fitness function (based on generating the state splitting tree [16] and thus considering all states of the FSM); (ii) it was evaluated only on relatively small FSMs; (iii) only completely specified machines were considered. In the work described in the present paper the fitness function is simpler and computationally easy to compute, and it also generated UIOs for partially specified machines.

3.1. Defining UIO generation as a search problem

When searching for a solution using GAs an efficient way must be defined to distinguish between potentially good and potentially bad solutions. A fitness function has been defined in order to represent the UIO sequence generation as a search problem. The fitness function determines how suitable a given transition sequence is to be a UIO sequence.

In order to verify if an input sequence would produce a UIO an algorithm with complexity is of $O(nl)$ has to be executed where n is the number of states of the FSM and l is the length of the input sequence. Instead the proposed fitness function has complexity of $O(l)$, and this fitness function aims to reward sequences that are likely to be UIOs. Picking a less computationally complex algorithm for the fitness function is important since the algorithm can be executed several times for each state.

A transition ranking process is completed first before the fitness function is ready to be used. This process ranks each input/output pair of the specification machine according to how many times it reoccurs in the transition table (a table with all the transitions of a machine) of the machine. A pair that occurs only once gets the lowest rank, a pair that occurs twice is ranked next etc. Pairs that have the same number of occurrences in the transition table get the same rank. For example Table 1 shows a ranked transition table for the FSM from Figure 1 ($M1$).

It is important to note that execution costs for different transitions are simply assumed to be equal. Also equally ranked pairs are assumed to have similar ability to construct valid UIOs. Where this does not hold it would be straightforward to introduce extra information into the fitness function without increasing the complexity of the algorithm.

The fitness algorithm used in this paper rewards a potential solution according to the ranks of the input/output pairs the sequence contains. The fitness function reflects the belief that the more lower ranked transitions a sequence contains, the more likely it is to define a UIO. Some reported experiments in Section 4 investigate this claim. In fact if there is an input/output pair that is unique, then it automatically becomes a UIO, identifying the state from which the transition initiates. This fitness function however does not account for infeasible test sequences if partially specified FSMs are considered. Input characters testing unspecified transitions could result in unexpected behaviour of the IUT. Hence

TABLE 1. Transition table for the FSM from Figure 1 with I/O rankings.

Start state	Input/output	End state	Rank
1	a/0	1	1
1	b/1	2	1
2	a/1	2	0
2	b/1	3	1
3	a/0	2	1
3	b/0	1	0

this fitness function works only for fully specified machine and in order for a partially specified machine to be used a completeness assumption has to be made. Below we explain how the fitness function can be adapted for partially specified machines.

Now consider the FSM $M1$ in Figure 1. Using the fitness function defined above the fitness of an input/output sequence would be the sum of the ranks assigned to the input/output pairs it is composed of. If we consider the sequence $a/0$, $b/1$ as a potential UIO for $s1$ of $M1$ and use the ranking provided on Table 1, a fitness value of 2 will be derived. This sequence is not a UIO as the same sequence could be executed and the same output observed from $s3$ of $M1$. On the other hand while the sequence $b/1$, $a/1$ is considered from $s1$ a fitness value of 1 could be derived. This sequence is a UIO and its fitness value reflects the higher chance of it being a UIO compared with the sequence before that.

Not all FSMs are completely specified and protocols systems are typically partially specified [16]. In strong conformance testing assumptions on how the non-core transitions are to be treated are made hence converting the machine into a completely specified FSM. For example one scenario is to add a transition with null output that stays in the same state. An alternative completeness assumption may be that if a transition is not in the core, then the machine makes a transition to an error state and outputs an error symbol. The missing transitions are treated as being 'do not cares' in weak conformance testing. The implementation is only required to have the same core behaviour and can be arbitrary or undefined for the missing transitions.

Further refinements to the fitness function allow it to work for partially specified machines. This could facilitate weak conformance testing without a completeness assumption. For the purpose a simulator of the specification FSM was constructed. The FSM simulator (λ^* and δ^*) is a lite version of the IUT that only determines if a test sequence is feasible from a given start state and if not it indicates how close it came of being feasible. If an input character from a sequence represents an infeasible transition from a given state the input is ignored by leaving the FSM in the same state and then the next input character in the sequence is considered. Hence the whole input sequence under consideration can be evaluated by the fitness function even when an infeasible

```
validValue := 0
strengthValue := 0
S_k := S_UIO
If (l = 0) then return φ
For(i := 1 to l) //for all the inputs/characters
   S_m := S_k
   y_i := λ(S_k, x_i)
   S_k := δ(S_k, x_i)
   If (S_k ≠ φ)
      //There is a transition with this input
      validValue := validValue + 1
      strengthValue := strengthValue + r_{S_k,x_i}
   EndIf
   Else
      strengthValue := strengthValue + penaltyValue
      S_k := S_m
   EndElse
EndFor
return l − validValue + strengthValue
```

FIGURE 4. UIO fitness algorithm.

character has been reached. The fitness of infeasible input sequences is penalized according to how close the sequence came to be valid, while valid sequences are not penalized at all. The algorithm for the fitness function proposed is presented in Figure 4. The parameters involved are as follows: S_{UIO} is the test state; x is a single input character; y is single output character; l represents the length of the input sequence; r is the set of transition rankings where $r_{s,x}$ represents the rank for the transition initiating from state s when input x is fed and *penaltyValue* is a penalty constant or function that penalizes the fitness when an unspecified transition is triggered.

The test sequence generated like this would enable strong conformance testing with a less restrictive completeness assumption and weak conformance testing without any completeness assumption for a partially specified machine.

The whole process of searching for a UIO for each state of a given FSM can be easily automated as only the transition table of the FSM is required.

A GA using this fitness function is directed towards generating input sequences that contain mostly input/output pairs with lower frequency in the transition table corresponding to feasible transitions (in the specification). The fitness function represents the search for a UIO sequence as a function minimization problem so an input sequence with a lower fitness value is considered to be more likely to form a UIO sequence since it is made up of more low ranked transitions.

3.2. Input sequence representation and GA

Generating a UIO sequence for a given state of an FSM would involve finding an appropriate input sequence that generates

a unique output sequence. A specification simulator of the FSM can be used to simulate a transition path and generate the corresponding output. Hence a genotype representing a potential solution for a given state will only need to encode an input sequence.

A phenotype representing a sequence of characters can easily be represented as a genotype made of chromosomes for each character. Then each character can be represented as it is or encoded in binary notation. As described in Section 2 the classic GA approach would be to encode the characters in binary, but both methods could be applied to this problem. The GA tool used for UIO generation [42] supported only binary, hence that was the method of choice. Also in an attempt to reduce premature convergence in the population Gray coding was used instead of standard binary encoding [38].

A type checking process could be used to discard genotypes that do not represent valid phenotypes. When using binary representation the information that it translates to should ideally be in increments of the power of 2. Hence an input for an FSM with binary alphabet can be presented with a single digit in binary, an input with input alphabet of size 4, with 2 binary digits etc. However there is a problem if the FSM considered has an input alphabet of size that is not a power of 2. In such cases a special type checking must be performed on all chromosomes within a genotype considered for fitness evaluation. The essence of this type checking is to ignore those binary combinations that do not translate to input characters from the input alphabet of the FSM considered. In the cases where a genotype is produced with invalid chromosome(s), the gene recombination, or generation in the case of the initial population generation, is repeated until a genotype where all the chromosomes are valid is produced.

This could potentially affect the speed of the algorithm as the input alphabet of the FSMs considered increases. However this was not evident in the experiments reported in Section 4. Sometimes the size of the input alphabet for an FSM is slightly bigger than its optimal binary representation (e.g. input alphabet of 17 will necessitate the use of a binary string of length 5 just because of one extra input character and introduces 15 redundant binary combinations). In such cases alternative binary to character translation techniques can be used [41] that distribute the number of valid characters and reduce the number of redundant binary combinations, but optimizing this part of the generation algorithm is not a focus of this paper.

The fitness function is designed so that it can compare only input sequences of the same length. For testing efficiency shorter sequences are more desirable, however we chose to separately consider the problem of having a fitness function and data representation that effectively addresses both the problem of UIO sequence generation and the length of such a sequence generated simultaneously.

Sequences of various lengths can be represented in binary for the GA in two ways. The first way is to simply have genotypes of different lengths encoding input sequence of different lengths. In this case the problem of how to apply the genetic recombination techniques has to be considered. Some work has been done on variable length genotype recombination, however these methods are very domain specific and no generic form is available [43]. A different approach is to encode different length input sequences by using the same length genotypes. This could be done by introducing a reset or sequence termination character to the sequence input alphabet. When such a character is reached in a sequence, the remaining characters encoded in the genotype will be ignored. In both situations the fitness function will favour shorter sequences to longer ones as they are likely to get a lower fitness value because of the fewer transitions involved. Initial experiments found that such a fitness function would always favour a single character sequence with just a reset character. Hence in order to generate a minimal UIO a set of generation attempts was made with gradually increasing sequence size.

3.3. Generating UIOs using GAs

After a fitness function and a phenotype representation technique are defined a GA can be used to find UIOs for all the states of an FSM. Verifying whether an input sequence is a valid UIO for a given state of an FSM is computation intense—$O(nl)$ for sequences of length l and n state FSM. So after a GA search stops, instead of checking if all the population individuals of the GA are UIOs only the sequence with the best fitness is considered. The result need not be a UIO sequence since not all FSMs have UIOs for all states or the GA might have converged prematurely i.e. the search might have converged to a local minimum. To increase the confidence that the input/output sequence found is the minimal length UIO for any given state i.e. a global minimum in the search space has been reached, the GA should be executed a number of times and only the best result kept.

For every GA execution the initial population is a set of randomly generated input sequences (genotypes). The corresponding output can be obtained from the FSM simulator generated from the FSM specifications (transition table). Each generated input sequence is type checked to see whether it represents a specified sequence of input of characters for the FSM under test. If not a new sequence is randomly generated until the initial population consists entirely of specified input sequences. The fitness is evaluated only for valid sequences. Hence any repeated attempts to generate a sequences are not counted as fitness evaluations. Starting the GA with a population of valid input sequences increases the probability of generating new valid input sequences after crossover. The crossover and mutation operators recombine the existing genotypes in such a way that input sequences representing specified transitions with lower ranked input/output pairs are rewarded. Input sequences that represent some unspecified

transitions, specified transitions with higher ranked input/output pairs or a combination of both would be rewarded less and penalized.

An example of how the GA recombination operators can help in this search follows. Lets consider an FSM $M2$ for which the sequence $a/1$, $b/0$, $c/1$ is a UIO for $s1$. Assuming that *abc* is the only minimal UIO for that state lets take *aab* and *cbc* as two potential solutions in the population of the GA searching to find that UIO. Recombining these two sequences using a crossover at the first point would generate the necessary solution *abc*. Alternatively a crossover at the second point would generate the sequence *aac* that after a mutation at the second point can again produce the required *abc*.

The GA for every search terminates either after a set number of recombinations or if the population gets saturated with the same solution and does not improve for a number of generations. The lowest possible value for the fitness function cannot be negative but otherwise is unknown. Hence the GA cannot be set to terminate after an optimal solution is found. The only exception is when a single input character represents a UIO, then the fitness value evaluates to 0 and the GA terminates. Hence the GA currently used might have generated a solution much earlier than it actually terminates, but we have not yet attempted to optimize this aspect of the GA. Further work will aim to improve the fitness function and the generation algorithm so that fewer GA cycles are necessary before a solution is found.

4. EXPERIMENTS

Most FSM examples available in the literature are not very large. A set of relatively small real FSM systems exists that is used for benchmarking purposes [44]. This set can be used to examine the effects of the UIO generation algorithm on small but real FSMs and in order to examine how it performs on larger FSMs a set of larger randomly generated FSMs was used.

The first set of experiments considers a set of 11 real FSMs (Table 2). The FSMs ranged in size from 4 to 27 states and 10–108 transitions. The second set of experiment was conducted on a set of 23 randomly generated FSMs (Table 3). These FSMs ranged from 5 to 360 states and 14–901 transitions in size.[1] Both sets consisted only of deterministic, strongly connected and minimal but not necessarily completely specified FSMs.

A breadth first search (BFS) algorithm can be used to enumerate through all possible input sequence combinations. By verifying each combination we can exhaustively (up to a fixed input sequence limit) find all the minimal length UIOs (within that limit). This approach would require each input sequence to be verified using a UIO verification algorithm ($O(nl)$). On the other hand the GA approach presented in this paper verifies only one input sequence at the end of a GA execution. For that reason it is difficult to present a precise comparison of effort between the GA and a BFS algorithm, but a rough figure biased towards the BFS is presented.

The minimal UIOs found for all the states of an FSM by the two GAs and random algorithm were considered. The shortest UIO for each state was listed. The longest UIO in this list was used as an indicator of what maximum length input sequence a BFS algorithm would be expected to generate for a given FSM in a worst case scenario. This figure

TABLE 2. List of the 11 real FSM examples used.

FSM	States	Transitions	Inputs	Outputs
dk15	4	32	8	11
mc	4	32	8	8
bbtas	6	24	4	4
beecount	7	51	8	4
dk14	7	56	8	15
dk27	7	14	2	3
shiftreg	8	16	2	2
dk17	8	32	4	5
lion9	9	25	4	2
dk512	15	30	2	4
dk16	27	108	4	5

TABLE 3. List of the 23 randomly generated FSM examples used.

FSM	States	Transitions	Inputs	Outputs
1	5	14	4	2
2	10	33	4	2
3	20	51	4	2
4	39	87	4	2
5	50	136	4	2
6	73	177	4	2
7	90	218	4	2
8	98	250	4	2
9	113	296	4	2
10	132	316	4	2
11	158	393	4	2
12	180	450	4	2
13	203	498	4	2
14	209	553	4	2
15	227	568	4	2
16	244	611	4	2
17	264	658	4	2
18	291	765	4	2
19	305	771	4	2
20	311	765	4	2
21	323	809	4	2
22	347	856	4	2
23	360	901	4	2

[1]The experiments were carried out on FSMs with at most 360 states due to the prototype tool being limited to FSMs with no more than 1000 transitions. This restriction was due to a combination of Java features and the tool design.

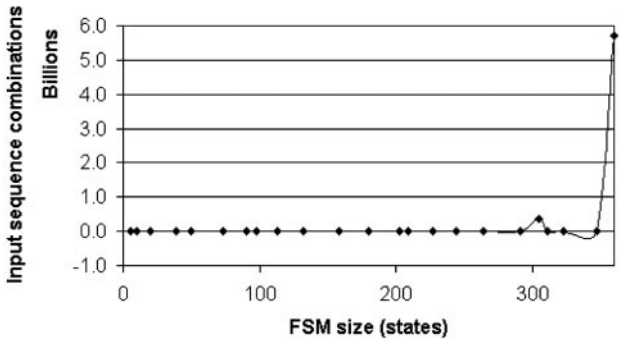

FIGURE 5. Difference in effort between worst case BFS and current GA results in attempt to find all UIOs of an FSM.

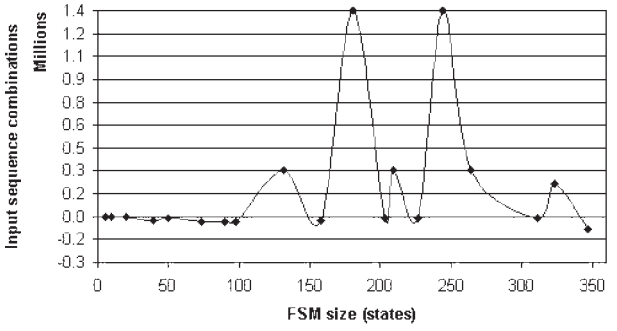

FIGURE 6. Difference in effort between worst case BFS and current GA results in attempt to find all UIOs of an FSM—four worst performing FSMs for BFS removed from graph.

is compared with the number of fitness evaluations (including unsuccessful UIO generation attempts) by the GAs and random UIO generation (the 2 GAs and random were given the same effort in terms of fitness evaluations). Figure 5 shows the *difference* between these two figures for all 23 randomly generated FSMs. As some of the BFS input sequence variations go into billions Figure 6 shows the same information but filtering the four worst estimated BFS effort FSMs. From the graph it is clear that for many FSMs the BFS algorithm could have been more efficient to use. This is because of the mainly short UIOs found for many of the FSMs. However the graphs also indicate that the BFS is much worse in some cases, where the GA performed well.

It was expected that when small FSMs are considered the real advantage of the new method cannot always be observed over the random test sequence generation method. However as the size of the FSMs considered increases the proposed method is expected to outperform the random method. Since BFS is not feasible for those FSMs where the benefits of using GA are likely to be observed, BFS was not included in the experiments.

Some results justifying the UIO generation algorithm choice are presented first. Then the actual performance of

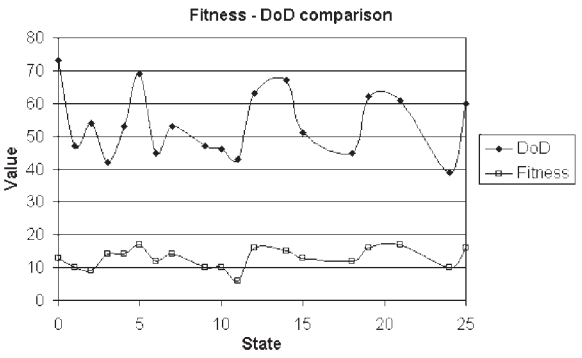

FIGURE 7. Fitness function value and DoD for a set of UIOs for FSM dk16.

the algorithm is compared with the random generation algorithm. The reason for using two different GA types was to experiment if the slightly different heuristics can generate better results. The first GA used a single point crossover and mutation while the second used a complex multiple point crossover and mutation. In general the second GA tended to find a solution slightly faster than the first GA, but they produced the same results. Hence for most FSMs the two GAs show identical performance.

4.1. UIO generation process

For any successful heuristic search it is imperative that a fitness function is selected that guides the search correctly towards a solution. In the search for UIOs the DoD metric can be used [30]. A DoD compares the output sequence β generated by an input sequence α from state s_i with the corresponding output sequence from state s_j. We can extend this notion and instead of comparing the output sequence of s_i only with that of s_j, where s_j is just another state, we can compare it with all states apart from s_i. We sum all the individual DoD values into one cumulative DoD for a given UIO. This process is of the same complexity as the UIO verification algorithm—$O(ln)$. In this paper we refer to this cumulative DoD value.

Figure 7 has two graphs representing the DoD and fitness values of 19 input sequences for the *dk*16 FSM, the largest from the set of real FSM examples. These 19 input sequences represented 19 UIOs for different states of that FSM. The vertical scale of the graph represents the fitness and DoD values while the horizontal represents the state of the UIO. It can be seen how the shape of the fitness function closely follows the shape of the DoD, except for the extent of the actual rises and falls of the DoD. This indicates that the fitness function, although not calculating the DoD for a given input sequence, can serve as a rough estimate of which input sequences are likely to have higher DoD and hence are likely to be UIOs. Therefore the fitness function

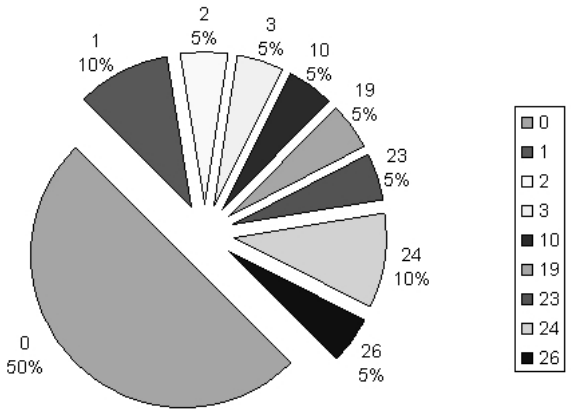

FIGURE 8. Positions in the GA population where the first valid UIO was found for each state of FSM dk16 (the largest of the real FSMs).

is likely to be directing the search in a positive direction without the full expense of calculating the DoD.

As mentioned before the UIO generation process used involved verifying whether a given input sequence is a UIO for a given state. After a GA has terminated only the highest ranked element in the final population is verified to see whether it represents a UIO because of the computational complexity involved with this checking process. Verifying an input sequence as a UIO is the most expensive part of the algorithm but it would not make sense to verify only the top ranked individual of a population if such individuals do not tend to be UIOs. Figure 8 represents the rank of the first element within the 20 terminated GA populations which generated UIOs for the *dk*16 FSM (the largest of the real FSMs). Half of the UIOs were found at the top, 0-th position of their corresponding GA population. The next highest ratio represented only 10% of the results. The rest of the real FSMs had even higher ratio of UIOs found at the top, 0-th position of their corresponding GA populations. This suggests that we lose little by verifying only the top ranked individual but we reduce the complexity of the whole UIO generation process since we repeat the search if a UIO is not found. It is simple to adapt the algorithm so it checks all elements of the final population or some fixed proportion of this.

4.2. UIO generation

A set of experiments involving UIO generation was run using the two sets of FSMs. Two slightly different GAs and a random search algorithm were used for every FSM. After each UIO generation attempt a simple algorithm was used to determine whether the sequence is indeed a valid UIO and does not contain unspecified transitions. The GAs used a single, ranked population where fitter genotypes are added by removing the genotypes with the lowest rank. The genotype selection was done using roulette wheel selection [34]. Gray coding [41] was used as the chromosome representation technique. The recombination operators used were uniform crossover and uniform binary mutation with mutation rate of 0.05. The first GA used the classic genotype recombination while the second GA used a chromosome recombination where each input character for a transition sequence is represented as a separate chromosome. The second GA performs recombination independently on each character of the input sequence. The termination criteria were population saturation or up to 10,000 fitness evaluations. A UIO generation attempt for a given state in the FSM involved no more than 3 GA executions, for each of the sequence sizes (number of chromosomes) considered with up to 25 inputs. The fittest phenotype after each GA termination was considered as a potential UIO sequence. As soon as a valid UIO was found for a given state in the FSM the search moved to the next state. For the randomly generated FSMs no more than 15 GA executions were considered for each sequence size up to 45 inputs because these FSMs are larger and we expected that more effort would be required to generate UIOs.

After sequences were generated with the 2 GAs, random sequence generation was applied. After a number of random input sequence generations, within the FSM input alphabet constraints, the sequences were ranked and the fittest one was checked to determine whether it was a UIO. The number of random generation attempts (to generate a UIO) for a state of the FSM used was equal to the average number of attempts it took the GAs to generate a UIO for that particular state. Every attempt to generate a sequence for a given state was repeated for sequence sizes ranging from the shortest to the longest UIO sequence found for this state by the GAs. The random search was given at least the same computational power in terms of number of fitness evaluations and UIO verification attempts.

Figure 9 and Table 4 show the results of the UIO generation algorithm conducted on the set of 11 real FSMs. For each FSM two different types of GA algorithms and a random generation algorithm were executed in an attempt to generate UIOs for each state.

Some of the FSMs considered have a very small number of states. For such FSMs a single input character might represent a UIO. In such cases it is obvious that the random algorithm will be effective. For example all the UIOs in the *mc* FSM were of length 1. It is also important to note that not all FSMs have UIOs for all states. For example the *lion*9 and *becount* FSMs have UIOs for only 2 of their states, and they were found by the UIO generation algorithms. The number of UIOs generated were compared with results reported in [45, 46]. FSMs *dk*14–17 and *dk*512 were reported to have the same UIO state coverage as we found. In [45] the *dk*16 FSM was reported to have UIOs for 21 of its 27 states, however [46] reported that it only has UIOs for 20 states and we manually verified that. The GA produced UIOs for these 20 states. FSMs *mc*, *bbtas* and *shiftreg* had UIOs

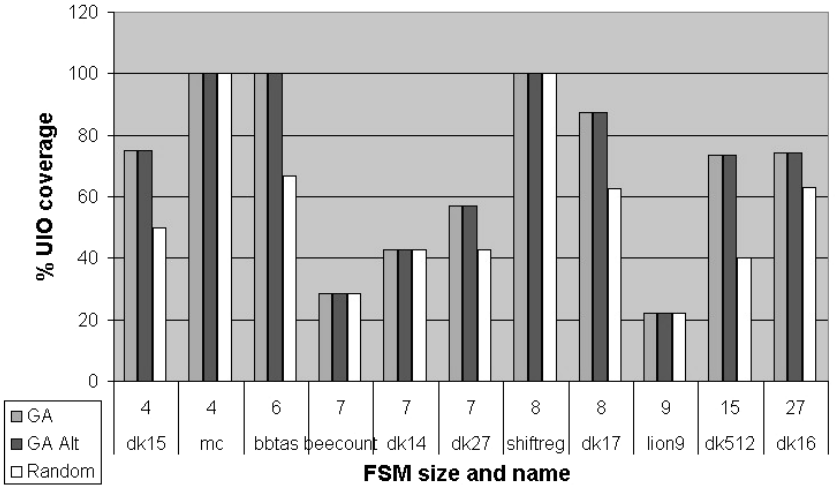

FIGURE 9. Percentage state coverage in UIOs generated by GA compared with random algorithm. Results for real FSMs.

TABLE 4. Percentage state coverage in UIOs generated by GA compared with random algorithm. Results for real FSMs.

FSM	States	GA %	GA Alt %	Ran. %	Diff. %
dk15	4	75	75	50	**25**
mc	4	100	100	100	0
bbtas	6	100	100	67	**33**
beecount	7	28	28	28	0
dk14	7	43	43	43	0
dk27	7	57	57	43	**14**
shiftreg	8	100	100	100	0
dk17	8	88	88	63	**25**
lion9	9	22	22	22	0
dk512	15	73	73	40	**33**
dk16	27	74	74	63	**11**

generated for all their states. This shows that for each FSM the GA UIO generation managed to find at least one UIO for all the states that had one. Also for most of the FSMs the GA-based UIO generation outperformed the random generation generating up to 33% better results. As expected not all the UIOs generated were minimal.

Now consider the experiments with (larger) randomly generated FSMs. Both GA search-based UIO generation techniques performed better for all 23 randomly generated FSMs, sometimes generating UIOs for up to 62% more states than the random search. The 2 GAs produced identical UIO state coverage results. Figure 10 shows the number of states for which a UIO has been generated as a percentage of the total number of states of the FSM using the three methods. Figure 11 shows the same data but plots the difference in the percentage between the random search and the two GA methods. Here it appears that the difference between the GA and random algorithm increases as the size of the FSMs increases. Both graphs clearly illustrate the potential

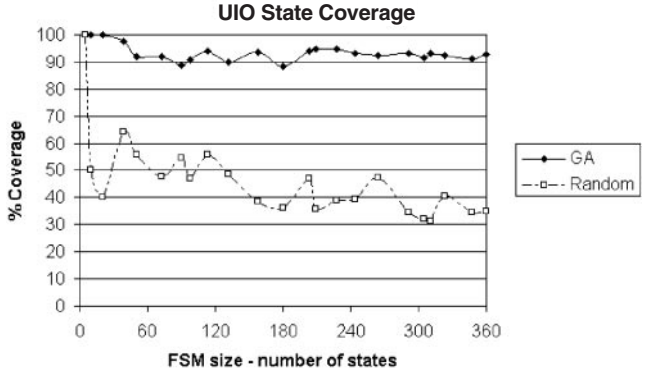

FIGURE 10. Percentage state coverage in UIOs generated by GA compared with random algorithm. The two GAs produced identical results. Results for randomly generated FSMs.

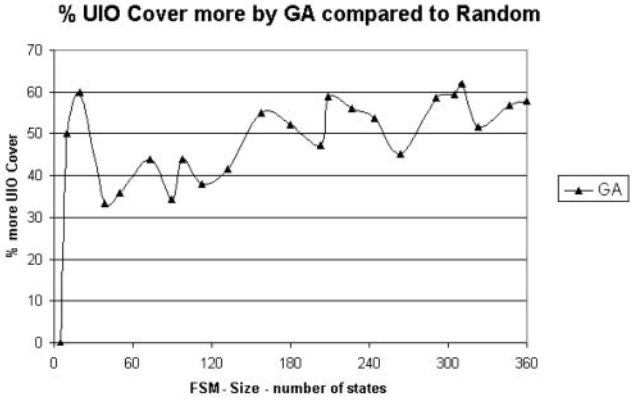

FIGURE 11. Percentage difference in UIOs generated by GA compared with random algorithm. Results for randomly generated FSMs.

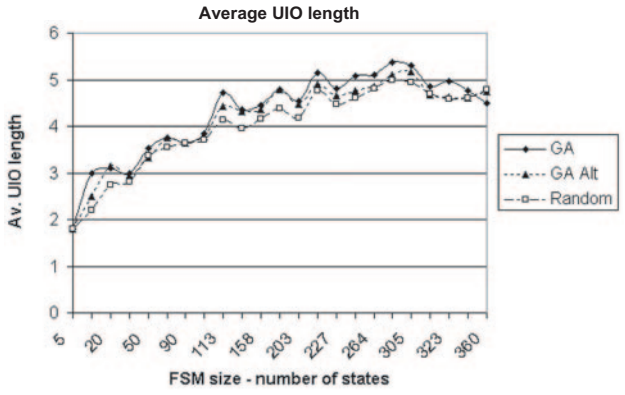

FIGURE 12. Average UIO size found for the randomly generated FSMs.

advantage of using GA search against random search for UIOs, when using the fitness function considered. It is important to remember that different FSMs have different properties. For example not all FSMs have UIO sequences for all their states. Hence the graphs are not very smooth. Again, not all the UIOs generated were minimal.

Another interesting result was that the average UIO sequence size was much shorter than expected as in the worst case the length of a UIO is exponential in terms of the number of states of the FSM [4]. In fact, most of the UIO sequences seem to be very short, even for larger FSMs. In comparison, a separating sequence is expected to be of size $n - 1$ at most, but it has been observed that its expected size is of $O(\log(n))$ [47]. Figure 12 shows the average UIO sequence length for each of the 23 FSMs using the GA methods and the random search. The graph does not seem to increase exponentially, but it actually seems to increase at a rate less than linear. Since most of the larger FSMs on the graph have state coverage as high as 95%, indicating that there are not many UIOs left to be found, it seems that most of the UIOs tend to be very short.

5. CONCLUSIONS

State verification is an important part of conformance testing for FSMs. UIO sequences are commonly used for state verification because of their advantages over the other methods. The problem of generating such sequences however is known to be NP-hard [16]. While a random algorithm could be used it does not always produce acceptable results. GAs have previously been used to generate UIOs for relative small and completely specified FSMs [18].

In this paper we define the problem of finding UIO sequences as a search problem. We define a computationally efficient fitness function of $O(l)$ complexity for an input sequence of size l that is used to guide a GA. UIOs for both completely and partially specified FSMs were generated. Our approach considers partially specified FSMs and generates UIOs that can also be used for weak conformance testing without completing the FSM.

We investigated the performance of a GA search for UIOs for an FSM using this fitness algorithm on a number of real and some larger randomly generated FSMs and report the results.

UIOs were computed using GA and random search. The experiment included two groups of FSMs: a set of 11 real FSM specifications of small size; and a set of 21 randomly generated FSMs with up to 360 states. The fitness function appears to direct the search towards generating UIOs. The experiments show that the GA outperforms (up to 62% better) or is at least as good as a random search for UIO sequences. As the size of the FSMs increased the difference between the performance of the GA and random UIO generation also increased.

The results also show that the average UIO size tends to be small even for larger FSMs. Most of the UIOs found were no longer than 10 input/output pairs. Searching for UIOs using a BFS algorithm for some of the larger FSMs considered could run into billions of input sequence generations in a worst case scenario (judging from the minimal UIOs we have found for those FSMs). However BFS could be more efficient than GA for shorter UIOs. This could suggest that BFS or even random search can be very useful for generating most of the UIOs, which are very short. GA search can subsequently be used to search for longer UIOs which are otherwise computationally difficult to identify using BFS.

REFERENCES

[1] Hennie, F. C. (1964) Fault-detecting experiments for sequential circuits. In *Proc. Fifth Annual Symp. Switching Circuit Theory and Logical Design*, Princeton, NJ, November, pp. 95–110.

[2] Chow, T. S. (1978) Testing software design modelled by finite state machines. *IEEE Trans. Softw. Eng.*, **4**, 178–187.

[3] Aho, A., Dahbura, A., Lee, D. and Uyar, M. U. (1991) An optimization technique for protocol conformance test generation based on UIO sequences and rural chinise postman tour. *IEEE Trans. Commun.*, **39**, 1604–1615.

[4] Lee, D. and Yannakakis, M. (1994) Testing finite state machines: state identification and verification. *IEEE Trans. Comput.*, **43**, 306–320.

[5] Sidhu, D. P. and Leung, T. K. (1989) Formal methods for protocol testing: a detailed study. *IEEE Trans. Softw. Eng.*, **15**, 413–426.

[6] Shen, Y., Lombardi, F. and Dahbura, T. (1989) Protocol conformance testing using multiple UIO sequences. In *Proc. IFIP WG6.1 9th Int. Symp. Protocol Specification Testing and Verification*, Amsterdam, Holland, pp. 131–144.

[7] Yang, B. and Ural, H. (1990) Protocol conformance test generation using multiple uio sequences with overlapping. In *Proc. ACM Symp. Communications Architectures & protocols (SIGCOMM '90)*, pp. 118–125.

[8] Miller, R. E. and Paul, S. (1993) On the generation of minimal-length conformance tests for communication protocols. *IEEE/ACM Trans. Netw.*, **1,** 116–129.

[9] Shen, X. and Li, G. (1992) A new protocol conformance test generation method and experimental results. *Proc. 1992 ACM/SIGAPP Symp. Applied Computing (SAC '92)*, pp. 75–84. ACM Press.

[10] Tanenbaum, A. S. (1996) *Computer Networks* (3rd edn). Prentice Hall, Upper Saddle River, NJ, USA.

[11] Gonenc, G. (1970) A method for the design of fault detection experiments. *IEEE Trans. Comput.*, **19**, 551–558.

[12] Ural, H., Wu, X. and Zhang, F. (1997) On minimizing the lengths of checking sequences. *IEEE Trans. Comput.*, **46**, 93–99.

[13] Hierons, R. M. and Ural, H. (2002) Reduced length checking sequences. *IEEE Trans. Comput.*, **51**, 1111–1117.

[14] Rezaki, A. and Ural, H. (1995) Construction of checking sequences based on characterization sets. *Comput. Commun.*, **18**, 911–920.

[15] Sabnani, K. K. and Dahbura, A. T. (1988) A protocol test generation procedure. *Comput. Netw. ISDN Syst.*, **15**, 285–297.

[16] Lee, D. and Yannakakis, M. (1996) Principles and methods of testing finite state machines—a survey. *Proc. IEEE*, **84**, 1090–1123.

[17] Goldberg, D. E. (1989) *Genetic Algorithms in Search, Optimisation and Machine Learning*. Addison-Wesley Publishing Company, Reading, MA.

[18] Guo, Q., Hierons, R. M., Harman, M. and Derderian, K. (2004) Computing unique input/output sequences using genetic algorithms. In *Proc. Formal Approaches to Software Testing: Third Int. Workshop, FATES 2003, LNCS* **2931**, pp. 169–184. Springer, New York.

[19] Jones, B. F., Eyres, D. E. and Sthamer, H.-H. (1998) A strategy for using genetic algorithms to automate branch and fault-based testing. *Comput. J.*, **41**, 98–107.

[20] Jones, B. F., Sthamer, H.-H. and Eyres, D. E. (1996) Automatic structural testing using genetic algorithms. *Softw. Eng. J.*, **11**, 299–306.

[21] Beizer, B. (1990) *Software Testing Technique* (2nd edn). Van Nostard Reinhold, New York.

[22] Petrenko, A., Yevtushenko, N. and v. Bochmann, G. (1996) Testing deterministic implementations from nondeterministic FSM specifications. In *Proc. IFIP TC6 9th Int. Workshop on Testing of Communicating Systems*, Darmstadt, Germany, September 9–11, pp. 125–140. Chapman & Hall, Ltd.

[23] Hwang, I., Kim, T., Hong, S. and Lee, J. (2001) Test selection for a nondeterministic FSM. *Comput. Commun.*, **24**, 1213–1223.

[24] Hierons, R. M. (2004) Testing from a non-deterministic finite state machine using adaptive state counting. *IEEE Trans. Comput.*, **53**, 1330–1342.

[25] Moore, E. (1956) Gedanken-experiments on sequential machines. *Automata Studies*, **34**, 129–153.

[26] Kohavi, Z. (1978) *Switching and Finite Automata Theory*. McGraw-Hill, New York.

[27] Luo, G., Petrenko, A. and von Bochmann, G. (1993) Selecting test sequences for partially-specified nondeterministic finite state machines. Technical Report IRO-864. Department d'Informatique et de Recherche Opérationnelle, Université de Montréal.

[28] Luo, G., von Bochmann, G. and Petrenko, A. (1994) Test selection based on communicating nondeterministic finite-state machines using a generalized WP-method. *IEEE Tran. Softw. Eng.*, **20**, 149–162.

[29] Inan, K. and Ural, H. (1999) Efficient checking sequences for testing finite state machines. *Inform. Softw. Technol.*, **41**, 799–812.

[30] Naik, K. (1995) Fault-tolerant UIO sequences in finite state machines. In *Proc. 8th IFIP Int. Workshop on Protocol Test Systems*, Evry, France, September, pp. 201–214.

[31] Son, H., Nyang, D., Lim, S., Park, J., Choe, Y.-H., Chin, B. and Song, J. (1998) An optimized test sequence satisfying the completeness criteria. In *Proc. 12th Int. Conf. Information Networking (ICOIN-12)*, Tokyo, Japan, January, pp. 621–625. IEEE.

[32] Wang, B. and Huthinson, D. (1987) Protocol testing techniques. *Comput. Commun.*, **10**, 79–87.

[33] Shen, Y., Scoggins, S. and Tang, A. (1991) An improved RCP-method for protocol test generation using backward UIO sequences. In *Proc. ACM Symp. Applied Computing (SAC 1991)*, Kansas City, MO, April, pp. 284–293. ACM Press, New York.

[34] Srinivas, M. and Patnaik, L. M. (1994) Genetic algorithms: a survey. *IEEE Comput.*, **27**, 17–27.

[35] Pargas, R. P., Harrold, M. J. and Peck, R. R. (1999) Test-data generation using genetic algorithms. *J. Softw. Test. Verif. Rel.*, **9**, 263–282.

[36] Michael, C. C., McGraw, G. and Schatz, M. A. (2001) Generating software test data by evolution. *IEEE Trans. Softw. Eng.*, **27**, 1085–1110.

[37] Tracey, N., Clark, J., Mander, K. and McDermid, J. (2000) Automated test-data generation for exception conditions. *Softw. Pract. Exper.*, **30**, 61–79.

[38] Whitley, D. (1999) A free lunch proof for gray versus binary encodings. In *Proc. Genetic and Evolutionary Computation Conf.*, Orlando, FL, USA, July, pp. 726–733. Morgan Kaufmann, CA, USA.

[39] Beasley, D., Bull, D. R. and Martin, R. R. (1993) An overview of genetic algorithms. Part 1: fundamentals. *Univ. Comput.*, **15**, 58–69.

[40] Beasley, D., Bull, D. R. and Martin, R. R. (1993) An overview of genetic algorithms. Part 2: research topics. *Univ. Comput.*, **15**, 170–181.

[41] Michalewicz, Z. (1996) *Genetic Algorithms + Data Structures = Evolution Programs* (3rd revised and extended edn). Springer-Verlag, Berlin, Heidelberg, New York.

[42] Derderian, K. (2002) *General Genetic Algorithm Tool*. Technical Report. Available at www.karnig.co.uk/ga/ggat.html.

[43] Goldberg, D., Deb, K. and Theirens, D. (1993) Toward a better understanding of mixing in genetic algorithms. *Soc. Instrum. Cont. Eng. J.*, **32**, 10–16.

[44] LGSynth91 (1991) *Logic synthesis and Optimization Benchmarks*. Technical Report 3, University of California. Available at www.ece.pdx.edu/polo/function/LGSynth91.

[45] Sun, D., Vinnakota, B. and Jiang, W. Fast state verification. In *Proc. 35th Annual Conf. Design Automation*, San Francisco, CA, USA, June, pp. 619–624. ACM Press, New York.

[46] Sun, H., Gao, M. and Liang, A. (2001) Study on UIO sequence generation for sequential machine's functional test. In *Proc. 4th Int. Conf. ASIC*, Shanghai, China, October 23–25, pp. 628–632. IEEE.

[47] Trakhtenbrot, B. A. and Barzdin, Y. M. (1973) *Finite Automata: Behavior and Synthesis.* North-Holland, Amsterdam.