An Improved Approach to Passive Testing of FSM-based Systems

Hasan Ural, Zhi Xu and Fan Zhang
SITE, University of Ottawa, Ottawa, Ontario, K1N 6N5, Canada
{ural, zxu061, fzhang}@site.uottawa.ca

Abstract

Fault detection is a fundamental part of passive testing which determines whether a system under test (SUT) is faulty by observing the input/output behavior of the SUT without interfering its normal operations. In this paper, we propose a new approach to Finite State Machine (FSM)-based passive fault detection which improves the performance of the approach in [4] and gathers more information during testing compared with the approach in [4]. The results of theoretical and experimental evaluations are reported.

1. Introduction

Passive fault detection is a testing technique used in fault management of a system under test by observing its input/output behaviors without interfering its normal operations [5]. In Finite State Machine (FSM)-based passive fault detection, the specification of the system under test (SUT) is modeled as an FSM M, SUT N is treated as a black-box FSM, and the tester wishes to determine whether N is faulty with respect to M by observing a sequence Q of I/O pairs from N where the starting state (when Q starts) of N is unknown. Such a decision can be based on the number of states that are compatible with Q. A state s of M is *compatible with* Q if Q is a trace of M starting at s. If the number of states compatible with Q is zero then Q is sufficient to determine that N is faulty. Otherwise, Q is insufficient to determine whether N is faulty. That is there are one or more states compatible with Q and Q needs to be augmented by an additional I/O sequence of N to continue with the fault detection.

Lee et al developed algorithms for FSM-based passive fault detection [4]. Their approach can be summarized as follows: suppose that the starting state of N is any state of M, check the observed sequence Q of I/O pairs one-by-one from the beginning, reduce the size of the set S' of possible current states by eliminating impossible states until either S' is empty (N is faulty) or there is at least one state in S' (no fault is detected by Q). This approach has been applied to FSM-based systems [10, 11, 12] and has been extended to systems specified in the Extended FSM model by [1, 2, 5, 6, 10] and to systems specified in the Communicating FSM model by [7, 8].

The algorithm in [4] is comprehensive but not efficient enough. In this algorithm, every state of M needs to be checked. However, the number of states compatible with Q is usually comparatively small and checking every state of M would be unnecessary. Further, this algorithm only determines the set of possible current states when it terminates. The information about possible starting state and possible trace corresponding to Q is not provided unless a post-processing is performed. Clearly, the approaches derived from [4] also have these two shortcomings. To improve the efficiency of FSM-based passive fault detection and gather more information during testing, we propose a new approach to FSM-based passive fault detection which is based on the following approach: randomly pick a state s in subset S_0 of the set of states of M and determine whether Q is the trace of M at s. If s is compatible with Q, stop and declare that Q is not sufficient to determine whether N is faulty. In this case, Q is a trace of M at s and the current state of M can be determined readily. Otherwise, continue to check other states in S_0. After checking all the states in S_0, if no state is found to be compatible with Q, then N is declared to be faulty. Note that we took S_0 to be equal to the set of states of M when we perform analytical and experimental comparisons of the approach we propose with the approach in [4] in an effort not to put the approach in [4] at a disadvantage.

The rest of the paper is organized as follows: Section 2 defines the terms and notations used in the paper. Section 3 describes algorithms for the proposed approach for the FSM-based passive fault detection, and compares the computational complexities of these algorithms. Section 4 presents the results of an experimental evaluation. Section 5 concludes the paper.

2. Preliminaries

An *FSM* M is a quintuple = $(S, X, Y, \delta, \lambda)$, where S

Second International Workshop on Automation of Software Test (AST'07)
0-7695-2971-2/07 $20.00 © 2007 IEEE

$= \{s_1, s_2, ..., s_n\}$ is a finite set of states with $n = |S|$ and $s_1 \in S$ as the *initial state*, X is a nonempty finite set of inputs, Y is a nonempty finite set of outputs, δ is a state transition function that maps $S \times X$ to S, and λ is an output function that maps $S \times X$ to Y. These two functions are extended to input sequences $I \in X^*$ in the standard manner. The FSM M defined above is *deterministic*, i.e., if for each input $x \in X$, there is at most one transition defined at each state of M.

M can be represented by a directed graph $G = (V, E)$ (Figure 1) where a set of vertices $V = \{v_1, v_2, ..., v_n\}$ represents the set of states of M and a set of edges $E = \{(v_j, v_k; x/y): v_j, v_k \in V\}$ represents all specified transitions of M, i.e., edge $e = (v_j, v_k; x/y)$ represents a state transition from state s_j to s_k with input $x \in X$ and output $y \in Y$, and the I/O pair x/y is the *label* of e. The label of a path $e_1 e_2 \ldots e_r$, $e_i \in E$, $1 \le i \le r$, is the concatenation of the labels of e_i and is called an I/O sequence. The I/O sequence $I/\lambda(s_i, I)$ is called the *trace of M at s_i*.

In this paper, we assume that both M and N are deterministic FSMs, an I/O sequence Q is observed from N, and a set of possible starting states S_0 of M is given. We wish to determine whether there is no state s in S_0 such that Q is a trace of M at s.

Example 1. Let M be as in Figure 1 and $S_0=\{s_1, s_2\}$. Q = "(a/0)(b/0)(a/0)(b/0)(a/0)(b/0)(b/1)" = "abababb/0000001". Since "0000001" = $\lambda(s_1$, abababb), Q is the trace of M at s_1. Thus, Q is declared to be insufficient to determine whether N is faulty. If Q = "(a/0)(b/0)(a/0)(b/0)(a/0)(b/0)(a/1)" = "abababa/0000001", Q is not a trace of M at any state, thus N can be reported to be faulty.

$Q = (x_1/y_1)(x_2/y_2)_{\ldots}(x_k/y_k)$ denotes an I/O sequence of length k. Q_j^p is the prefix of Q of length j, Q_j^s is the suffix of Q of length k-j, $1 \le j \le k$.

3. Algorithms for Passive Fault Detection

We first present the algorithm proposed by Lee et al [4], then propose three new algorithms for FSM-based passive fault detection. In order to make the analysis and further comparisons of the algorithms, we consider the *number of comparisons* between the actual output y_j and the expected output $\lambda(s, x_j)$ as the measure of computational complexity, $1 \le j \le k$, $s \in S$.

3.1 The Approach of Lee et al in [4]

In order to facilitate comparisons, we have rewritten the algorithm given in [4] as Algorithm 0 without changing its computational complexity.

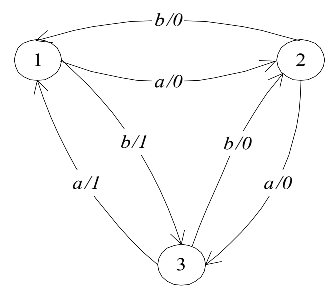

Figure 1. An FSM *M*

Algorithm 0
Given: FSM $M = (S, X, Y, \delta, \lambda)$, $S_0 = S$,
I/O sequence $Q = (x_1/y_1)(x_2/y_2)_{\ldots}(x_k/y_k)$
Begin:
$j \leftarrow 1$; /* j is the counter for I/O pairs */
$S' \leftarrow S_0$;
while ($j \le k$)
if ($S' \ne \emptyset$) {
$S'' \leftarrow \emptyset$;
for (each $s \in S'$) /* check each state in S'*/
if ($y_j = \lambda(s, x_j)$) $S'' \leftarrow S'' \cup \delta(s, x_j)$;
/*redundant states in S'' are removed*/
endfor
$S' \leftarrow S''$, $j \leftarrow j+1$;
}
else /* $S' = \emptyset$ */
return ("N is faulty");
endwhile
return ("No fault is detected by Q and
the set of possible current states is S'");
End

If the while loop terminates before the entire Q is checked, N is declared to be faulty. Otherwise, Q is declared to be insufficient to determine whether N is faulty. In this case, the possible current states are determined but the possible starting states (where Q starts) are unknown. In order to find the set of possible starting states, a post-processing will be needed.

Theorem 1 (Lee et al [4]) Let S_j denote the set of possible current states right after the first j I/O pairs of Q, i.e., $S_j = \delta(S_0, x_1 x_2 \ldots x_j)$. The computational complexity of Algorithm 0 is $C_1 = \sum_{j=1}^{k} | S_{j-1} |$.

Proof: In Algorithm 0, every state in the set of possible current states will be checked to compare its related I/O pair with the current I/O pair in Q, i.e., for a state s in S_j, there will be one comparison between y_{j+1} and the expected output $\lambda(s, x_{j+1})$, and $| S_j |$ comparisons are needed to check the set S_j. Thus, the total number of comparisons is $\sum_{j=1}^{k} | S_{j-1} |$.

Example 2. Applying Algorithm 0 to M (Figure 1) and

Q = "abababb/0000001", the number of comparisons is $\sum_{j=1}^{k} |S_{j-1}| = 3 + 2 + 2 + 2 + 2 + 2 + 2 = 15$.

3.2 The Proposed Approach

Algorithm 1 is based on our proposed approach which checks, for each state $s \in S_0 \subseteq S$, whether Q is a trace of M at s. It terminates when Q is verified to be a trace of M at a state $s \in S_0$ or when all states in S_0 are checked and no state is found compatible with Q.

Algorithm 1

```
Given:  FSM M = (S, X, Y, δ, λ), S0 ⊆ S,
        I/O sequence Q = (x1/y1) (x2/y2)...(xk/yk)
Begin:
    i ← 1;      /* i is the state counter */
    while (i ≤ n)
        j ← 1; /* j is the counter for I/O pairs */
        s ← si;
        /* s will represent δ(si, x1... xj-1) when j > 1 */
        while (j < k AND yj = λ(s, xj))
            s ← δ(s, xj); j ← j+1;
            /*s is updated as the current state*/
        endwhile
        if (j = k AND yj = λ(s, xj))
            return ("Q is a trace of M at state si and
                    the possible current state is s");
        else
            i ← i +1;
    endwhile
    return ("N is faulty");
End
```

Algorithm 1 either declares N to be faulty or yields both the possible current state (s) and possible starting state (s_i) once a state compatible with Q is found.

Theorem 2 For the given state s_i of M and I/O sequence $Q = x_1 \ldots x_k / y_1 \ldots y_k$, let $c_i(Q)$ denote the largest number j ($1 \le j \le$ k) such that $y_1 \ldots y_{j-1} = \lambda(s_i, x_1 \ldots x_{j-1})$ and $\lambda(\delta(s_i, x_1 \ldots x_{j-1}), x_j) \ne y_j$. Let $C_{2worst}(M, S_0, Q) = \sum_{i=1}^{n} c_i(Q)$. Let $C_2(M, S_0, Q)$ denote the computational complexity of Algorithm 1. If s_r is the first state of M such that Q is a trace of M at s_r, then $C_2(M, S_0, Q) = \sum_{i=1}^{r} c_i(Q)$; if N is faulty, then $C_2(M, S_0, Q) = C_{2worst}(M, S_0, Q) = \sum_{i=1}^{n} c_i(Q)$.

Proof: In Algorithm 1, each state in S_0 is checked to determine whether it is compatible with Q. The checking procedure for a state s_i will not stop until it confronts a mismatch (then the next state s_{i+1} will be selected to check); or the entire sequence Q has been checked and no mismatch found (then s_i is reported to be compatible with Q). The whole checking procedure will terminate when a state compatible with Q is found or when all the states have been checked and no state is found to be compatible with Q. Assume $y_1 \ldots y_{j-1} = \lambda(s_i, x_1 \ldots x_{j-1})$ but $y_j \ne \lambda(\delta(s_i, x_1 \ldots x_{j-1}), x_j)$, it means j comparisons (denoted by $c_i(Q)$) are needed to determine that Q is not a trace of M at s_i. If $s_r \in S_0$ is the first state of M such that Q is the trace of M at s_r, Algorithm 1 will detect mismatch in checking $s_1 \ldots s_{r-1}$ and stop after checking s_r. Thus, the total number of comparisons needed is $\sum_{i=1}^{r} c_i(Q)$.

Example 3. Applying Algorithm 1 to M (Figure 1) and Q = "abababb/0000001", with $r = 1$, the number of comparisons is $C_2(M, S_0, Q) = \sum_{i=1}^{r} c_i(Q) = c_1(Q) = 7$.

Algorithm 1 is simple and straightforward, however, it encounters the *redundant checking problem* which is: two traces starting from different states converge to the same state after applying Q_j^p. In Algorithm 1, the common part Q_j^s will be rechecked redundantly. In contrast, Algorithm 0 avoids the redundant checking problem by removing redundant states in the set of possible current states.

Algorithm 2 below attempts to combine the merits of both Algorithm 0 and Algorithm 1. Let S_j denote the set of possible current states right after the first j I/O pairs of Q, i.e., $S_j = \delta(S_0, x_1 x_2 \ldots x_j)$. In Algorithm 2, Algorithm 0 is used first to reduce the size of S_j and then Algorithm 1 is used on the current S_j with the remaining portion of the I/O sequence Q.

Algorithm 2

```
Given:  FSM M = (S, X, Y, δ, λ), S0 ⊆ S, q ≤ k,
        I/O sequence Q = (x1/y1)(x2/y2)...(xk/yk)
Begin:
    j ← 1;  /* j is the counter for I/O pairs */
    S′ ← S0;
    while (j ≤ q)  /* Algorithm 0 up to Qq^p */
        if (S′ ≠ ∅) {
            S″ ← ∅;
            for (every s ∈ S′ )
                if (yj = λ(s, xj))  S″ ← S″ ∪ δ(s, xj);
                /*redundant states in S″ are removed*/
            endfor
            S′ ← S″; j ← j+1;
        }
        else    /* S′ = ∅ */
            return ("N is faulty");
    endwhile
    while (S′ ≠ ∅)
        /* Algorithm 1 starting from Sq = S′ */
        randomly choose a state s from S′;
        S′ ← S′\{s};  j ← q +1;
        /* j is the counter for I/O pairs */
        while (j < k AND yj = λ(s, xj))
```

$s \leftarrow \delta(s, x_j)$; $j \leftarrow j+1$;
/* s is updated as the current state */
endwhile
if (($j = k$ AND $y_k = \lambda(s, x_k)$)
return ("No fault is detected by Q and the possible current state is s");
endwhile
return ("N is faulty");
End

The computational complexity of Algorithm 2 is obtained by combining the results in Theorem 1 and Theorem 2 and it is affected by the selection of a value for variable q. Suppose that, in Algorithm 0 part of Algorithm 2, j_1 is the minimum number of I/O pairs needed to eliminate the redundant checking problem and j_2 is the minimum number of I/O pairs needed to reduce the size of possible current states to be one or zero. Obviously, j_1 is smaller than j_2. Clearly,

if $q < j_1$, the redundant checking problem remains;

if $q > j_2$, Algorithm 2 is the same as Algorithm 0;

if $j_1 \le q < j_2$, Algorithm 2's performance is at least equal to that of Algorithm 0. However, since j_1 and j_2 are both determined by Q and M, it is difficult to determine a proper value for q. Also, Algorithm 0 part makes Algorithm 2 unable to determine the possible starting state unless a post-processing is performed.

Algorithm 0 is not efficient enough as it has to check all the states in S_0 and doesn't provide information about possible starting states. Algorithm 1's efficiency is influenced by the redundant checking problem. Algorithm 2 attempts to eliminate the redundant checking problem by combining the merits of both Algorithm 0 and Algorithm 1 but the effort is limited as it introduces a variable q for which it is difficult to find an appropriate value. Also, Algorithm 2 cannot determine the possible starting state. Algorithm 3 presented below overcomes the drawbacks of these three algorithms:

Algorithm 3
Given: FSM $M = (S, X, Y, \delta, \lambda)$, $S_0 \subseteq S$,
I/O sequence $Q = (x_1/y_1)\,(x_2/y_2)\ldots(x_k/y_k)$
Begin:
$F_{1\ldots k\text{-}1} \leftarrow \varnothing$; $i \leftarrow 1$;
while ($i \le n$)
$j \leftarrow 1$; /* j is the counter for I/O pairs */
$s \leftarrow s_i$;
/* s will represent $\delta(s_i, x_{1\ldots}\, x_{j\text{-}1})$ when $j > 1$ */
while ($j < k$ AND $y_j = \lambda(s, x_j)$)
$s \leftarrow \delta(s, x_j)$;
/* s is updated as the current state */
if ($s \in F_j$) /* to eliminate *redundant checking problem* */
break; /* state s has already been checked. Thus, end this trace */
else
$j \leftarrow j + 1$;
endwhile
if ($j = k$ AND $y_j = \lambda(s, x_j)$)
return ("Q is a trace of M at state s_i and the possible current state is s");
else
$i \leftarrow i + 1$;
/*record the trace*/
if ($j > 1$) **add** $\delta(s_i, x_1\ldots x_l)$ **to** F_l, l=1,…,j-1;
endwhile
return ("N is faulty");
End

The data structure $F_{1\ldots\, k\text{-}1}$ is used to record the tracing history and therefore to avoid the redundant checking problem. If $\delta(s_i, x_1x_2\ldots x_j) \in F_j$, $1 \le j \le k$, then $\lambda(\delta(s_i, x_1x_2\ldots x_j), x_{j+1}\ldots x_k)$ has already been checked and $y_{j+1\ldots}\, y_k \ne \lambda(\delta(s_i, x_1x_2\ldots x_j), x_{j+1}\ldots x_k)$. So, Q_j^s will not need to be checked and the checking started from state s_i will stop. After checking a state s_i, if s_i is not an eligible starting state, Algorithm 3 adds the trace history starting from s_i into $F_{1\ldots j\text{-}1}$. If s_i is compatible with Q, the possible starting state s_i and its corresponding trace are determined.

Theorem 3 For a given state s_i of M and an I/O sequence $Q = x_1\ldots x_k/y_1\ldots y_k$, let $c'_i(Q)$ denote the largest number j ($1 \le j \le k$) such that (1) $y_1\ldots y_{j\text{-}1} = \lambda(s_i, x_1\ldots x_{j\text{-}1})$ and $\lambda(\delta(s_i, x_1\ldots x_{j\text{-}1}), x_j) \ne y_j$.; (2) for every l ($1 \le l \le j$-1), $\delta(s_i, x_1\ldots x_l) \notin F_l$. If the r^{th} state checked, s_r, is the first state of M such that Q is a trace of M at s_r, then the computational complexity of Algorithm 3:

$C_3(M, S_0, Q) = \sum_{i=1}^{r} c'_i(Q)$;

if N is faulty, then $C_{3\text{worst}}(M, S_0, Q) = \sum_{i=1}^{n} c'_i(Q)$;

if $r = 1$, $C_{3\text{best}}(M, S_0, Q) = c'_1(Q)$.

Proof: Compared to Algorithm 1, the checking procedure of Algorithm 3 on state s_i will stop when it encounters a mismatch with Q, the whole Q has been checked compatible, or $\delta(s_i, x_1x_2\ldots x_j) \in F_j$. Similar to Theorem 2, let $c'_i(Q)$ denote the largest number j ($1 \le j \le k$) before checking on s_i terminates. If s_r is the first state of M such that Q is the trace of M at s_r, then the total number of comparisons needed is $\sum_{i=1}^{r} c'_i(Q)$.

As the states compatible with Q are randomly dispersed in S_0, the process of sequentially checking the states within S_0 until a state compatible with Q is found can be modeled as the *Sampling without Replacement Model* [3].

Theorem 4 According to the *Sampling without Replacement Model*, assume that there are m states in S_0 which are compatible with Q, and r ($1 \le r \le n$–m+1) states are randomly selected from S_0. Then, the probability that the r^{th} state is the first state which is checked to be compatible with Q is given by

$$P_r(m)=\begin{cases}\dfrac{m}{n} & (r=1);\\ \dfrac{m(n-m)(n-m+1)\ldots(n-m-r+2)}{n(n-1)\ldots(n-r+1)} & (2\le r\le n-m+1).\end{cases}$$

Proof: Each different arrangement of states selected from S_0 is called a *permutation.* Suppose that r states are selected one at a time and removed from S_0 ($1 \le r \le n–m+1$). Then each possible outcome of this selection will be a permutation of r states from S_0, and the total number of these permutations will be $P_{n,r} = n(n–1)\ldots(n–r+1)$ [3]. $P_{n,r}$ is called the *number of permutations of n elements taken r at a time*. Thus, if $r=1$, the number of permutations is n; if $2 \le r \le n–m+1$, the number of permutations, in which the r^{th} state is the first state compatible with Q, is $mP_{n-m,r-1} = m(n–m)(n–m-1)\ldots(n–m–r+2)$. Then the probability of permutation that the r^{th} state is the first state compatible with Q is: if $r=1$, $P_r(m)= m/ P_{n,r} = m/n$;
if $2 \le r \le n\text{-}m+1$, $P_r(m) = mP_{n-m,r-1}/P_{n,r} = \dfrac{m(n-m)(n-m+1)\ldots(n-m-r+2)}{n(n-1)\ldots(n-r+1)}$.

Theorem 5 Suppose there are m ($0 \le m \le n$) states in S_0 which are compatible with Q. Let $P_r(m)$ denote the probability that the r^{th} state is the first state which is compatible with Q. The average computational complexity of Algorithm 3 is $A_3 =$

$$\sum_{r=1}^{n-m+1} P_r(m)C_3(M,S_0,Q)=\sum_{\mathrm{r}=1}^{\mathrm{n-m+1}}(P_r(m)\sum_{i=1}^{r}c_i'(Q)).$$

Proof: The average computational complexity of Algorithm 3 is the sum of the number of comparisons multiplied by its corresponding probability [9].

Example 4. Assume $n = 4$, $m = 1$, when $r = 1$, $C_3 = 4$; when $r = 2$, $C_3 = 5$; when $r = 3$, $C_3 = 7$; when $r = 4$, $C_3 = 9$ where C_3 stands for $C_3(M, S_0, Q)$; Then,
$A_3 =1 + 5/4 + 7/4 + 9/4 = 25/4$ (see Table 1).

Table 1. Average computational complexity analysis

r	C_3	$P_r\,(m{=}1)$	$P_r\,(m{=}1)\,C_3$
1	4	1/4	1
2	5	1∗3 / 4∗3 = 1/4	5/4
3	7	1∗3∗2 / 4∗3∗2 = 1/4	7/4
4	9	1∗3∗2∗1 / 4∗3∗2∗1 = 1/4	9/4

In general, the number of states compatible with Q may be zero, or more. If it is zero, it means that none of the states in S_0 is compatible with Q and N is faulty; if it is one or more than one, it means that the given Q is insufficient to determine whether N is faulty.

The general case can be simplified to the case in which the number of states which are compatible with Q is either one or zero. This stems from the fact that the essence of passive fault detection is to detect the existence of faults in N. If there is one or more states compatible with Q, it implies that the given Q is insufficient to come to a conclusion. Additional I/O sequence ΔQ is needed to continue with the fault detection. Thus, let $Q' = Q + \Delta Q$ denote the I/O sequence concatenating Q to ΔQ. Thus, the new set (M, S_0, Q') contains at most one state that is compatible with Q'. Let $P_r(m = 1)$ denote the probability that there is one compatible state in S_0 and it appears at the r^{th} ($1 \le r \le n$) selection. So, $P_r(m = 1) = 1/n$.

Theorem 6: If there is only one state in S_0 that is compatible with Q, the average computational complexity of Algorithm 3 is

$$A_3=\frac{1}{n}\sum_{r=1}^{n}C_3(M,S_0,Q)=\frac{1}{n}\sum_{\mathrm{r}=1}^{\mathrm{n}}(\sum_{i=1}^{r}c_i'(Q)).$$

Proof: The average computational complexity of Algorithm 3 is the sum of the number of comparisons multiplied by its corresponding probability.

3.3 Comparison of the Algorithms

The computational complexities of the three algorithms given in the previous subsections are summarized in Table 2.

Table 2. Computational complexity

Type of algorithm	Computational complexity
Algorithm 0	$C_1 = \sum_{j=1}^{k} \lvert S_{j-1} \rvert$
Algorithm 1	$C_2(M, S_0, Q) = \sum_{i=1}^{r} c_i(Q)$
Algorithm 3	$C_3(M, S_0, Q) = \sum_{i=1}^{r} c_i'(Q)$

- k is the length of Q, $|S_j|$ is the number of states in the set of possible current states,
- r is the number of states checked before a state compatible with Q is found,
- $c_i(Q)$ is the largest number j ($1\le j \le k$) such that $y_1\ldots y_{j-1} = \lambda(s_i, x_1\ldots x_{j-1})$ and $\lambda(\delta(s_i, x_1\ldots x_{j-1}), x_j) \ne y_j$
- $c'_i(Q)$ is the largest number j ($1\le j \le k$) such that $y_1\ldots y_{j-1} = \lambda(s_i, x_1\ldots x_{j-1})$ and $\lambda(\delta(s_i, x_1\ldots x_{j-1}), x_j) \ne y_j$ after eliminating the redundant checking problem

Below, we compare the computational complexities of Algorithm 0 and Algorithm 3 when the number of states in S_0 which are compatible with Q is one or zero. In Algorithm 0, once the set (M, S_0, Q) is fixed, the number of comparisons needed is determined and does not change during its application. On the other hand, the performance of Algorithm 3 is affected by the number of states in S_0 which are compatible with Q (see Theorem 4). Algorithm 0 and Algorithm 3 represent different perspectives on tracing order in passive fault detection. Algorithm 0 checks all the states in the set of possible current states with one I/O pair in Q at a time whereas Algorithm 3 selects one starting state from S_0 and exhausts all the possible transitions starting from this state according to the I/O sequence Q. If there is no state in S_0 which is compatible with Q, both Algorithm 0 and Algorithm 3 need to check the entire trace from every state in S_0 and thus they perform equally. That is $\sum_{i=1}^{n} c_i'(Q) = \sum_{j=1}^{k} | S_{j-1} |$. If there is only one state s in S_0 which is compatible with Q, then the total number of comparisons made by Algorithm 0 is $\sum_{j=1}^{k} | S_{j-1} |$ whereas the total number of comparisons made by Algorithm 3 is $\sum_{i=1}^{r} c_i'(Q)$ $(r \leq n)$. Clearly, $\sum_{i=1}^{r} c_i'(Q) \leq \sum_{i=1}^{n} c_i'(Q) = \sum_{j=1}^{k} | S_{j-1} |$, $(r \leq n)$ where the r^{th} state checked is the state compatible with Q.

Thus, Algorithm 3 always performs at least as well as Algorithm 0. The equality in their computational complexities occurs when $r = n$.

Based on the computational complexities of the algorithms presented above, several assertions can be made on their performance in different conditions.

When N is not determined to be faulty:

i) if there is no redundant checking problem,
   the performance of Algorithm 1 will be the same as that Algorithm 3 and be at least equal to that of Algorithm 0; and the performance of Algorithm 2 will be between those of Algorithms 0 and 1.

ii) if there is redundant checking problem,
   the performance of Algorithm 3 will be at least equal to those of Algorithms 0 and 1; and
   it is not possible to compare the performances of Algorithm 0, Algorithm 1, and Algorithm 2 analytically due to the redundant checking problem.

When N is determined to be faulty:

i) if there is no redundant checking problem, the performances of all the algorithms will be the same.

ii) if there is redundant checking problem,
   the performance of Algorithm 3 will be equal to that of Algorithm 0; the performances of Algorithms 1 and 2 will at most be equal to that of Algorithm 0; and it is not possible to compare the performances of Algorithms 0 and 2 analytically.

4. Experimental Evaluation

An experimental evaluation is made to compare the average computational complexity of the algorithms and to verify the validity of the assertions drawn above when $m = 0$ or 1. In the experiment, we use a set of randomly generated FSMs. This set consists of FSMs with different number of states ($|S_0| = |S|$), set X of inputs and set Y of outputs. We select 5 configurations in the form of ($|S_0|$, $|X|$, $|Y|$), namely (5, 3, 3), (10, 4, 4), (15, 4, 4), (20, 5, 5), (30, 10, 10). For each configuration, we generate 5 FSMs correspondingly. For each FSM M, two cases are considered.

In Case I, called *correct implementation*, there is exactly only one state in S_0 that is compatible with Q ($m = 1$). In Case II, called *faulty implementation*, there is no state in S_0 that is compatible with Q ($m = 0$) and "faulty" is expected to be reported. We create a faulty specification M' from M by altering either the output or next state of a (randomly) selected transition. In Case I (Case II), for every state s of M (M'), we generate three random I/O sequences of length $|S_0|*|X|*2$, $|S_0|*|X|*4$, $|S_0|*|X|*10$ respectively, starting from s; and when generating each I/O sequence Q, we randomly select a transition of the current state of M (M') and repeat this at the next state.

Then, we apply all four algorithms to the FSMs in these two cases and record the results. Table 3 shows the number of comparisons (between the actual output y_j and the expected output $\lambda(s, x_j)$), $1 \leq j \leq k$, $s \in S$, for each of the four algorithms. We see from Table 3 that,

- Algorithm 1, in Case I, has better performance than Algorithm 0 in average case and best case, but not in worst case. Also, in Case II, Algorithm 1 cannot beat Algorithm 0;
- Algorithm 2 performs the same as the Algorithm 0 because the number of states in the set of possible current states shrinks to one or zero in the "Algorithm 0" part of Algorithm 2.
- Algorithm 3, in Case I, needs fewer comparisons to find the compatible state and performs better than Algorithm 0; while in Case II, these two algorithms perform the same.

Experimental results confirm the assertions we present in Section 3 and show that Algorithm 3 performs best among these four algorithms when there is one state in S_0 compatible with Q (Case I).

5. Conclusions

In this paper, we proposed a new approach to Finite State Machine-based passive fault detection. Compared with the former approach in [4], the proposed approach (Algorithm 3) has better performance and provides more information during testing. Specifically, Algorithm 3 provides more information about possible starting state and possible trace compatible with the observed sequence Q and performs better in situations where there is only one state in S_0 that is compatible with Q. The results of both theoretical and experimental evaluations confirm this improvement over the approach in [4].

Table 3. Experimental results

			Case I : $m = 1$			Case II : $m = 0$		
Algorithm 0	$\|Q\|$	$\|S_0\|$	best	worst	average	best	worst	average
	60	5	64	74	65.5	6	58	17.8
	160	10	169	175	171.3	11	162	31.7
	240	15	254	262	258.0	16	252	42.8
	400	20	419	429	423.0	21	399	44.7
	1200	30	1229	1236	1231.9	31	1122	73.0
Algorithm 1	$\|Q\|$	$\|S_0\|$	best	worst	average	best	worst	average
	60	5	60	74	62.3	7	58	18.7
	160	10	160	172	165.5	11	202	36.9
	240	15	240	263	247.7	16	252	46.1
	400	20	400	432	409.3	21	744	52.9
	1200	30	1200	1235	1215.0	31	1531	83.7
Algorithm 2 $q = 5$ (q is defined in Algorithm 2)	$\|Q\|$	$\|S_0\|$	Best	worst	average	best	worst	average
	60	5	64	74	65.5	6	58	17.8
	160	10	169	175	171.3	11	162	31.7
	240	15	254	262	258.0	16	252	42.8
	400	20	419	429	423.0	21	399	44.7
	1200	30	1229	1236	1231.9	31	1122	73.0
Algorithm 3	$\|Q\|$	$\|S_0\|$	Best	worst	average	best	worst	average
	60	5	60	65	61.9	6	58	17.8
	160	10	160	171	164.3	11	162	31.7
	240	15	240	262	247.9	16	252	42.8
	400	20	400	427	409.6	21	399	44.7
	1200	30	1200	1234	1215.3	31	1122	73.0

6. Acknowledgments

This work was supported in part by grants from Natural Sciences and Engineering Research Council of Canada, and Ontario Centers of Excellence.

References

[1] B. Alcalde, A. Cavalli, D. Chen, D. Khuu and D. Lee (2004) “Network Protocol System Passive Testing for Faulty Management - a Backward Checking Approach,” *Proc. of IFIP FORTE’04*, *LNCS,* vol. 3235, pp.150-166.

[2] D. Chen, J. Wu, and T.L. Chu (2003) “An Enhanced Passive Testing Tool for Network Protocols,” *Proc. of ICCNMC’03*, pp.513-516.

[3] M.H. DeGroot and M.J. Schervish. *Probability and Statistics*. Boston: Addison-Wesley, 2002.

[4] D. Lee, A.N. Netravali, K.K. Sabnani, B. Sugla, and A. John (1997) “Passive Testing and Applications to Network Management,” *Proc. of ICNP’97*, pp.113-122.

[5] D. Lee, D. Chen, R. Hao, R.E. Miller, J. Wu and X. Yin (2002) “A Formal Approach for Passive Testing of Protocol Data Portions,” *Proc. of ICNP’02*, pp.122-131.

[6] D. Lee, D. Chen, R. Hao, R.E. Miller, J. Wu and X. Yin (2006) “Network Protocol System Monitoring – A Formal Approach with Passive Testing,” *IEEE/ACM Transactions on Networking*, vol.14, pp.424-437.

[7] R.E. Miller (1998) “Passive Testing of Networks Using a CFSM Specification,” *Proc. of IPCCC’98*, pp.111-116.

[8] R.E. Miller and K.A. Arisha (2001) “Fault Identification in Networks by Passive Testing,” *Proc. of 34th Annual Simulation Symposium*, pp.277-284.

[9] R. Neapolitan and K. Naimipour. *Foundations of Algorithms Using C++ Pseudocode, 3rd Edition*. Sudbury, Mass.: Jones & Bartlett Publishers, 2003.

[10] M. Tabourier and A. Cavalli (1999) “Passive testing and application to the GSM-MAP protocol,” *Information and Software Technology*, vol. 41, pp.813-821.

[11] J. Wu, Y. Zhao, and X. Yin (2001) “From Active to Passive: Progress in Testing of Internet Routing Protocols,” *Proc. of FORTE’ 01*, pp.101-118.

[12] Y. Zhao, X. Yin, and J. Wu (2001) “OnLine Test System, an Application of Passive Testing in Routing Protocols,” *Proc. of ICN’01*, pp.190-195.