A Stochastic Combinatorial Optimization Model for Test Sequence Optimization

Shuai Wang
1. Department of Automation, Tsinghua University
2. Tsinghua National Laboratory for Information Science and Technology, Tsinghua University, Beijing, China
wangshuai81@gmail.com

Yindong Ji[1,2], Shiyuan Yang[1]
1. Department of Automation, Tsinghua University
2. Tsinghua National Laboratory for Information Science and Technology, Tsinghua University, Beijing, China
{jyd, ysy-dau}@tsinghua.edu.cn

***Abstract*—Traditional FSM (finite state machine) based test sequence generation methods have three problems: 1) fake test results may occur; 2) unnecessary repetitive tests may exist; 3) actual test coverage rate could be low. These problems are mainly because of the dependences existing between transitions of test sequences. In this paper, to solve these problems, we defined a stochastic combinatorial optimization model to describe the test sequence generation problem from the dynamic viewpoint. Meanwhile, a recursive algorithm is proposed to give one optimal solution for the test sequence generation. This algorithm uses the weighted finite state machine model for the software being tested. At each test decision time, a test sequence will be generated from this model. After the execution of one test sequence and fault detecting, the weight value of this model will be updated. Simulation results show that the effective test efficiency and test coverage rate are evidently increased using our method. Especially, the fake test results are much less than transitional methods.**

***Index Terms*—stochastic combinatorial optimization model; test sequence optimization; test efficiency; test coverage rate**

I. Introduction

Where software can be modeled as a finite state machine (FSM), testing can be taken as checking the output value of several sequences of input values. Such sequences are called test sequences, which are usually derived from its FSM model of specification. Usually an input is given at an input port and the outputs associated with the input can be observed at the output ports. The outputs that are generated by the implementation of the software will then be compared with the expected outputs corresponding with the input.

Commonly, a test sequence can be divided into three parts: a preamble sequence, the transition to be tested, and a postamble sequence. The preamble sequence is a sequence of inputs which can lead the software to the head state of the transition to be tested. The postamble sequence is a sequence of inputs verifying the tail state of the transition to be tested.

Finite state machine based test has been widely applied in where a system is state-based, communication systems, protocol interoperability testing, protocol conformance testing, etc.. And a considerable amount of works have been devoted to test generation [1-6]. Several optimization methods for test sequence generation have been proposed [7-9]. The focus of these methods lies in how to combine fewer test sequences to fulfill test criteria. In the following test execution process, the execution is carried out according to this fix set of test sequences. Unfortunately, the test execution with fix test sequence set will cause the following problems.

1) The effective test efficiency is low, because there may exist unnecessary repetitive tests in the test execution process.

2) The actual test coverage rate is low because of the faulty premable sequence or postamble sequence. The transition to be tested may not be executed and checked at its expected conditions or even not be executed at all.

3) Fake test results may exist, because the fualty transitions in the premable sequence or the postmable sequence may interfere with the judgement of the transition to be tested.

These problems are mainly because of the dependences existing between transitions of test sequence. This means that transitions which have been executed may affect the testing and the judgments of final test results, even though they are not the desired transitions to be tested. That is, if the transitions in the preamble sequence and postamble sequence of a test sequence are wrong, the transition to be tested will be assigned a "fail" verdict even though it has been implemented correctly.

In order to solve these problems, several approaches have been designed. The abstract test case relation model (ATCRM) was proposed by Chanson and Li, which dynamically generated test cases [10]. However, it is harder to localize errors because the unit to be tested is a sequence of transitions rather than one transition. Moreover, the number of tests is large because the test set generated by ACTRM which consists of all possible paths from the initial state covers all possible behaviors of the implementation under test.

Another method called dynamic conformance test method (DCTM) was proposed by Myungchul Kim, Sangjo Yoo and et al. [11]. This method dynamically selects a test sequence using the test sequence tree, which is a data structure that dynamically represents all test sequences for a transition. In other words, it considers dependencies between transitions in order to give a more correct verdict to be tested. The advantages of DCTM compared to the method ATCRM are that number of tests is smaller with the same fault coverage and the ability to localize errors is improved because this model focuses on certain transition instead of test paths. But its fault detection strategy may still generate a plenty of tests.

In this paper, considering the effects of the faulty transitions in preamble sequence and postamble sequence and the dependences between transitions in test sequence, we propose a dynamic finite state machine (DFSM) method. In this method, the software under test is modeled as a finite machine assigned with weight value and Boolean value for each transition. The weight value and Boolean value may be changed during test execution process. They are seen as parameter vector for FSM model. The test sequence will be dynamically generated at every decision time based on this model for the transition with least test cost. When each test execution is over, the parameter vector of FSM will be update based on the execution result of this test and the history test data.

According to the simulation results in Section V, the advantages of DFSM compared with the method DCTM are:

1) The cost of tests is smaller than DCTM, when given the same fault coverage.
2) The DCTM method selects test sequences using test sequence tree. This tree represents all possible test sequences for a transition, so its construction and maintenance is a heavy work. Using our method, we only need to maintain a dynamic FSM for software under test, and it’s easy to achieve.
3) The DCTM method simply supposes that the faulty transition will be detected after all the test sequences for it are carried out. This usually cannot be fulfilled in practice. Usually, after a serial of test sequences for a transition are carried out, a diagnostic candidate set is generated, which usually contains several diagnostic candidates not a unique diagnostic result. With this consideration, we use the faulty sub-sequence to demonstrate the diagnostic result and the fault information for the following test selection.

The rest of this paper is organized as follows. The traditional FSM based test methods and the problems of testing with the fix test set are discussed in Section II. In section III a stochastic combinatorial optimization model is introduced to describe the test sequence generation problem after considering the dependences between transitions. The recursive algorithm based on dynamic finite state machine is discussed in Section IV to give an optimal solution to the test sequence generation. The simulation experiments for TCP test and comparison with transitional method and related work are given in Section V. Finally, the conclusion is presented in Section VI.

II. Test with FSM and Problem Analysis

Given an implementation I and a deterministic finite state machine (FSM) M that models the required behaviors of I, it is important to check I against M. For the sake of convenience, we will first recall the basic idea of testing with FSM, and then we will introduce the fault model of FSM. At the end of this section, an example will be discussed to demonstrate the problems when testing with fix test sequence set.

A. Test with FSM

Usually the software under test can be modeled by FSM that produces outputs on its state transitions after receiving inputs.

Definition 1: A finite state machine (FSM) is a five-tuple $DFA = (Q, \Sigma, \delta, q_0, \lambda)$ [12]

1) Q is the set of finite states;
2) Σ is the set of finite events which contains both the input events and output events, $\Sigma = I \bigcup O$;
3) $\delta : Q \times \Sigma \to Q$ are the state transition functions;
4) q_0 is the initial state;
5) $\lambda : Q \times \Sigma \to \Sigma$ are the output functions.

When the machine is in a current state $q \in Q$ and receives an input a, it moves to the next state specified by $\delta(q,a)$ and produces an output given by $\lambda(q,a)$.

A FSM can be represented by a directed graph $G = (V, E)$, where the set $V = \{v_1, \cdots v_n\}$ of vertices represents the set of specified states Q of the FSM and a directed edge represents a transition from one state to another in the FSM. An edge in G is fully specified by a triple $(v_i, v_j; L)$, where $L \equiv a_k / o_l$, $L^{(i)} \equiv a_k$ and $L^{(o)} \equiv o_l$. In this paper, it is assumed that G is strongly connected.

The approach taken in this paper for checking I against M is to test the implementation for the correctness of every specified transition of I. The procedure for testing a specified transition from state q_i to state q_j with input/output a_k / o_l takes place in three steps.

1) The implementation is leaded into state q_i
2) Input a_k is applied and the output is checked to verify that whether it is o_l, as expected, or not.
3) The new state of implementation is checked to verify that if it is q_j, as expected, or not.

Suppose that there exist a reset action which is applied to make the software return to its initial state. This ensures that each test is applied in the same state of the IUT. The reset action might involve some sequences of inputs or a single action such as a reset, or the system being closed off and then powered on again. We also suppose that M has the following feature, called a status message. For each state $q_i \in Q$, this message denotes the state uniquely, such as the unique input/output (UIO) sequence [6]. The tail state is verified by checking this message.

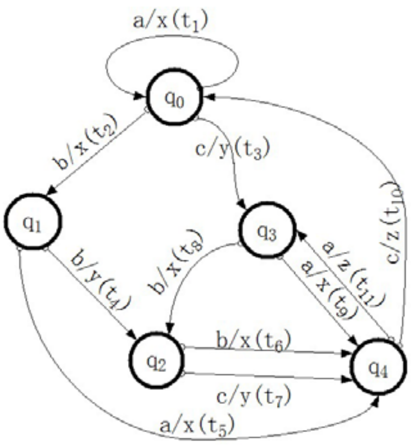

Figure 1. An example of finite state machine

In this paper, the test sequence for testing transition $(q_i, q_j; a_k / o_l)$, is constructed based on the U-method introduced in [13] as follows:

1) Constructing the reset action r to I so that I can be reset to the initial state.
2) Generating the shortest transition sequence that can lead the machine from the state q_0 to the state q_i.
3) Applying the input a_k which can enable the transition to be tested.
4) Generating the tail state q_j verifying sequence.

For the M with the status message feature, the test sequence for each transition in I is of the form

$$r_i; ts_{pre}; t; ts_{post} \tag{1}$$

where r_i is the reset action, ts_{pre} is the preamble sequence for the transition t, $t: q_j = \delta(q_i, a_k)$ is the transition to be tested, ts_{post} is the postamble sequence to check the tail state of transition t.

Example 1. An example FSM is shown in Fig. 1. By using the above method, one test sequence for transition $t_8: q_2 = \delta(q_3, b)$ is $r; t_3, t_8, t_6, t_{11}$, where t_3 is the preamble sequence that can lead the machine to the q_3, t_8 is the transition to be tested, t_6, t_{11} is the postamble sequence. From the model, we can see that q_2 is the unique state after the input sequence b, a is applied the outputs x, z are observed, so $b/x, a/z$ is one UIO sequence for state q_2.

B. Fault model

In the area of test generation for software testing, all existing generation methods should ensure full fault coverage, so fault model serves as a basis for test generation. A general survey on a variety of fault models in testing was given in [14]. The behavior of software under test can be formally defined as:

$$trace = \{I_1 / O_1, I_2 / O_2, \cdots, I_n / O_n\}. \tag{2}$$

We use $M = (Q, \Sigma, \delta, q_0, \lambda)$ stands for the required behaviors of software and $I = (Q', \Sigma', \delta', q_0, \lambda')$ stands for the implementation behaviors of software. If

TABLE I.
UNIQUE INPUT/OUTPUT SEQUENCES FOR EACH STATE

State	UIO sequence	State	UIO sequence
q_0	$b/x, b/y$	q_3	$a/x, c/z$
q_1	b/y	q_4	a/z
q_2	$c/y, a/z$		

$M =_{trace} I$, we say that I has no fault with respect to M; and if $M \neq_{trace} I$, we define the faulty types as follows:

1) Output fault: We say that a transition in I has a output fault if there exists M' such that $M' =_{trace} I$, and M' can be obtained from M by changing the output of certain transition.
$$q_i = q_i' \wedge \delta(q_i, a) = \delta'(q_i', a) \wedge \lambda(q_i, a) \neq \lambda'(q_i', a).$$
2) Transfer fault: We say that I has a transfer fault if there exists M', such that $M' =_{trace} I$ and M' can be obtained from M by changing the ending state of certain transition.
$$q_i = q_i' \wedge \delta(q_i, a) \neq \delta'(q_i', a) \wedge \lambda(q_i, a) = \lambda'(q_i', a).$$

Based on these two faults, for a complete defined FSM, there exist $((n-1)(q-1)+1)^{np-1}(n-1)(q-1)$ types faulty situation. This is very large number, so fault diagnosis for FSM is a difficult work.

C. Example analysis

With the method discussed in section II.A, we generate the full set of test sequences for every transition in the machine shown in Fig.1. The UIO sequence for each state is shown in Table I.

The generated sequences are in Table II.

The test tree is shown in Fig. 2.

Among the test sequences in Table II, if test sequence for transition i is included in test sequence j, the testing for the transition i can be performed by the test sequence

TABLE II.
TEST SEQUENCES FOR EACH TRANSITION

	Test aim	**Input**	**Expected output**	**Transition sequence**
ts_1	$t_1: q_0 \to q_0$	r, a, b, b	x, x, y	t_1, t_2, t_4
ts_2	$t_2: q_0 \to q_1$	r, b, b	x, y	t_2, t_4
ts_3	$t_3: q_0 \to q_3$	r, c, a, c	y, x, z	t_3, t_9, t_{10}
ts_4	$t_4: q_1 \to q_2$	r, b, b, c, a	x, y, y, z	t_2, t_4, t_7, t_{11}
ts_5	$t_5: q_1 \to q_4$	r, b, a, a	x, x, z	t_2, t_5, t_{11}
ts_6	$t_6: q_2 \to q_4$	r, b, b, b, a	x, y, x, z	t_2, t_4, t_6, t_{11}
ts_7	$t_7: q_2 \to q_4$	r, b, b, c, a	x, y, y, z	t_2, t_4, t_7, t_{11}
ts_8	$t_8: q_3 \to q_2$	r, c, b, c, a	y, x, y, z	t_3, t_8, t_7, t_{11}
ts_9	$t_9: q_3 \to q_4$	r, c, a, a	y, x, z	t_3, t_9, t_{11}
ts_{10}	$t_{10}: q_4 \to q_0$	$r, c, a,$ c, b, b	$y, x, z, x,$ y	$t_3, t_9, t_{10},$ t_2, t_4
ts_{11}	$t_{11}: q_4 \to q_3$	$r, c, a, a,$ a, c	$y, x, z, x,$ z	$t_3, t_9, t_{11},$ t_9, t_{10}

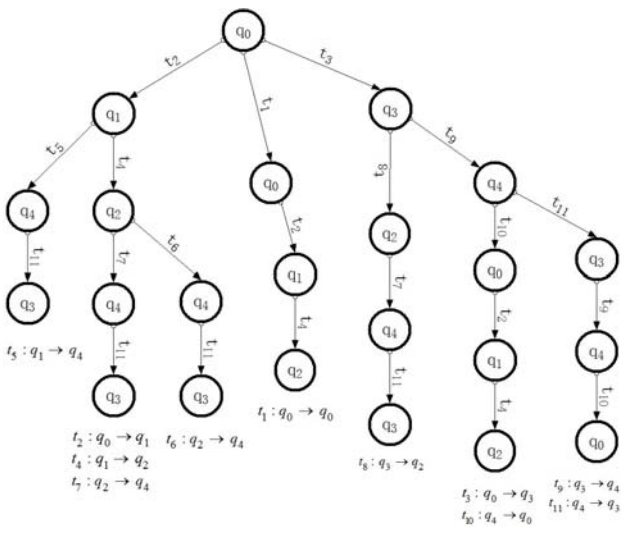
$t_5 : q_1 \rightarrow q_4$
$t_2 : q_0 \rightarrow q_1$
$t_4 : q_1 \rightarrow q_2$
$t_7 : q_2 \rightarrow q_4$
$t_6 : q_2 \rightarrow q_4$
$t_1 : q_0 \rightarrow q_0$
$t_8 : q_3 \rightarrow q_2$
$t_3 : q_0 \rightarrow q_3$
$t_{10} : q_4 \rightarrow q_0$
$t_9 : q_3 \rightarrow q_4$
$t_{11} : q_4 \rightarrow q_3$

Figure 2. Test tree of finite state machine

j as well. This inclusion relation is directly shown in Fig. 2. The test sequences shown in Table II are reduced to 7 test sequences shown in Fig. 2.

Suppose that the implementation has a faulty transition t_4 (e.g., generating the output 'z' for the input 'b' or the ending state of this transition is q_4). In this situation, when we execute the fix test sequence set shown in Fig. 2, it may cause the following three problems.

1) The fake test results occur. Not only will the test for the transition t_4 be assigned a "fail" verdict, transitions t_2, t_6, t_7 will also be assigned "fail" verdicts. This will occur even though they were implemented correctly. t_4 is one of the transitions in the preamble sequence for t_6 and t_7 , and t_4 is one of the transitions in the postamble sequence for t_2 . As a result, correct test results may not be produced.
2) The unnecessary repetitive tests exist. The test sequence t_2, t_4, t_6, t_{11} carried out for transition t_6 is unnecessarily, because it contains faulty transition t_4 . As a result, the "fail" test verdict is assigned to t_6 even though it was correctly implemented.
3) Actual test coverage rate is low. Some of transitions are not executed acutually, such as t_6 . Because of the fault of t_4 , the ending state after the execution of t_4 may not be q_2 , so t_6 may not be executed under the test purpose in test design phase. This causes the actual test coverage rate to be low.

III. STOCHASTIC COMBINATORIAL OPTIMIZATION MODEL

The traditional test method with FSM and its problems were discussed in last section. From the discussion we can see that with fix test sequence set, the faulty transitions in the preamble sequences and the postamble sequences may interfere with the execution and verification of the transitions to be tested.

Traditional test sequence generating methods and sequence optimization methods only concern the static effect of each test sequence and generate fix sequence set for testing. In order to solve the above problems, we consider the test sequence generation and optimization as a dynamic process. The result of each test sequence execution becomes an uncertain outcome because of the dependence between transitions in preamble and postamble sequences and the transition to be tested. Base on this idea, we construct the stochastic combinatorial optimization model for test sequence optimization in this section.

A. Uniform test generation and optimization model for traditional methods

Both our method and traditional methods can be seen as a programming problem, combining fewest test cost for the software under test. In order to compare the difference between our method and traditional methods, we first give the uniform definition of formal model for traditional methods.

This is achieved by the following hypotheses.

Hypothesis 1: Test sequences are executed one by one, so time is discrete, and in one step, one test is executed.

Hypothesis 2: Test sequence selection also happens at discrete time.

The first test sequence is applied to the software under test at time $t = 0$. The *k*th test sequence is applied to the software under test at time $t = k - 1$. $T = \{0, 1, 2, \cdots\}$ are also test decision time. At each time, one test decision is made and one test sequence is generated. The test sequences generated at each time compose the optimal test sequence set for the software.

In order to formally define this test decision process, we define the following basic concepts.

Definition 2 Test sequence: one executable state transition (input/output) path is defined as one test sequence for software. All test sequences compose the test sequence set Ts ,

$$Ts = \{ts_1, ts_2, \cdots, ts_n\}$$

where n is the number of test sequences.

Definition 3 Test target space: the test target space is determined by the test coverage rule. In this paper, the target space is defined as the transition set. We assume that the implementation of software contains *m* transitions. Then the initial target space is

$$T_0 = \{t_1 \cdots t_m\} .$$

The value of the test target space at time $t = i$ is the transitions which have not been tested.

$$T_i = \{t_i \cdots t_j, i \in [1, m], j \in [1, m], i \neq j\} . \quad (3)$$

Definition 4 Test Decision Space: selecting one test sequence at time *i* is defined as one decision. The value of the decision is defined as the test sequence selected.

$$d_i = ts_j (ts_j \in Ts, j \in [1, n]) . \quad (4)$$

All executable test sequences compose the decision space.

$$D = \{Ts_1, Ts_2, \ldots Ts_n\} . \quad (5)$$

Definition 5 Test Revenue Function: it is a function which is defined to compute the benefits that are get from the execution of one test decision. In other words, its value is the satisfaction level for test target.

$$R_i = r(d_i), i \in [1,n] . \tag{6}$$

In this paper, the value of R_i is the transition which have been tested through the executing of d_i.

$R_i = r(d_i) = \{t_{i1}, t_{i2}, \ldots t_{il}\} \bigcap T_i, i \in [1,n]$.

Definition 6 Test Cost Function: this function is defined to compute the cost generated from the execution of d_i.

$$C_i = c(d_i), i \in [1,n] . \tag{7}$$

Where C_i can stand for the money cost, time cost or other cost of the executing of d_i. In this paper, the value of C_i is weight sum of the transitions which are contained in d_i.

$$C_i = c(d_i) = sum(w(t_{i,j})) |, i \in [1,n], j \in [1, |d_i|] \tag{8}$$

where $|d_i|$ means the number of transitions contained in d_i.

Definition 7 Objective Function: f_1 is defined as the quantity of the transitions which have been tested. f_2 is defined as overall test cost generated from the execution of history test decisions.

$$\begin{aligned} f_1 &= \bigcup_{i=1}^{\tau} R_i = \bigcup_{i=1}^{\tau} r(d_i); \\ f_2 &= \sum_{i=1}^{\tau} C_i = \sum_{i=1}^{\tau} c(d_i). \end{aligned} \tag{9}$$

τ is the test ending time.

From $t=1$ to $t=\tau$, all test decisions compose a test decision sequence:

$$D = (d_1, d_2, \ldots d_\tau) . \tag{10}$$

Up to now, we get the test sequence set with the value of decision sequence which can fulfill the test coverage rule:

$Ts = (ts_i, ts_m, \ldots ts_\tau)$.

Usually, when we carry out test, we want to fulfill test coverage rule with least test cost. So, the test optimization objective is finding the optimal D to make:

$$(f_1 = T_0) \wedge \min(f_2) . \tag{11}$$

The optimization model of traditional methods which only consider the static effects of each test sequence can be formally defined as:

$$Model ::= (T, D, f_1, f_2) . \tag{12}$$

With this consideration, the test tree in Fig. 2 is one optimal set of test sequence for FSM in Fig. 1.

B. Stochastic combinatorial optimization model

If considering the dependences between transitions, the revenue and cost of one test sequence will become uncertain, because at the test beginning we cannot get the faults happening position and fault types. So the test revenue function and test cost function are redefined as:

$R_i' = r'(d_i) = a_i(f) r(d_i)$,

$C_i' = c'(d_i) = b_i(f) c(d_i)$,

where $a(f), b(f)$ are defined as the stochastic vectors in probability space (Ω, F, P).

Take the test sequence ts_4 for example. With traditional method, $R = r'(ts_4) = \{t_2, t_4, t_7\}$, $C = c'(ts_4) = 4$. Using our method, if t_2 is wrong, $R' = \varnothing$; if t_4 is wrong, $R' = \varnothing$; if t_7 is wrong, $R' = \{t_2\}$; if t_{11} is wrong, $R' = \{t_2\}$.

Then the optimization model for the test sequence generating is redefined as a stochastic combinatorial optimization model:

$$\begin{aligned} &Model ::= (T, D, f'_1, f_2', F); \\ &f_1' = \bigcup_{i=1}^{\tau} R_i' = \bigcup_{i=1}^{\tau} r'(d_i) = \bigcup_{i=1}^{\tau} a_i(f)(\{t_{i1}, t_{i2}, \ldots t_{il}\} \bigcap T_i) \\ &f_2' = \sum_{i=1}^{\tau} C_i' = \sum_{i=1}^{\tau} c'(d_i) = \sum_{i=1}^{\tau} b_i(f) c(d_i); \\ &\max(f_1'); \\ &\min(f_2'). \end{aligned} \tag{13}$$

where, $F, a(f), b(f)$ are defined as the stochastic vectors in probability space (Ω, F, P).

Note: Because of the faults existing in preamble sequences, some transitions may become unreachable, so the full coverage may not be achieved. At this situation, we use $\max(f_1')$ to stand for trying the best to fulfill the test coverage aim.

IV. TEST SEQUENCE GENERATION

In practice, the fault probability and fault type of each transition cannot be known before testing. So we cannot solve this stochastic combinatorial optimization model directly.

In this paper, a recursive algorithm is proposed to generate the optimal test sequences. The core idea of our method is combining the test executing process and test verifying process as a close loop. Different from previous work [10-11], we will not generate all test sequences at the beginning. At each test decision time, one test sequence will be generated. After one test execution, we will get more detailed knowledge on the implementation and reconfigure its FSM model. We name this dynamic finite state machine (DFSM) method. In DCTM method, it supposes that the faulty transition can be detected after all test sequences for one transition are carried out, but this cannot usually be met even though with single fault assumption[15-17] which is usually also not true. Considering this, DFSM uses the faulty sub-sequence to describe the fault information after one test sequence execution. This makes our method have more wide application areas. In this section we will first introduce the basic fault detection process after the execution of test sequence and then we will describe the test sequence generating process using DFSM method.

A. Fault detection process

The faulty transition information can be detected with the execution of one test sequence and test history data. The test analysis process may be a six-step process.

Step1: Generation of the expected outputs

As discussed in section II, a test sequence ts_i consists of m_i transitions $t_1, t_2, \cdots, t_{m_i}$. Its corresponding sequence of expected outputs is written as $o_1, o_2, \cdots, o_{m_i}$, where o_i is expected after transition t_i happening.

Step2: Generation of the observed outputs

After applying test sequence to the implementation of software, a corresponding observed sequence is written as $\hat{o}_1, \hat{o}_2, \cdots, \hat{o}_{m_i}$ for each test sequence ts_i.

Step3: Generation of the symptoms

Compare observed outputs with expected ones and identify all symptoms. Any difference $o_j \neq \hat{o}_j$ represents a symptom that is denoted as $s_{i,j}$. The symptom set for test sequence ts_i execution is

$S_i = \{s_{i,1}, s_{i,2}, \cdots s_{i,m}\}$.

Step4: Generation of the conflict set

For each symptom $s_{i,j} \mid o_j \neq \hat{o}_j$ of test sequence ts_i, determine its corresponding conflict set

$$FC_{i,j} = \{ fc_{j,m1}, \cdots fc_{j,mi} \}, \tag{14}$$

where, $fc_{j,ml}$ can be single transition with either output fault or transfer fault, and it also can be the combination of faulty transitions. A conflict set for a given symptom is defined to be the set of components which are supposed to be faulty to explain the generation of this symptom.

Step5: Generation of the diagnostic candidates

Diagnostic candidates are components which are suspected to be faulty to explain the observation of test execution ts_i. This is achieved by three steps:

5A) Computing the initial diagnostic candidate set "IDS" of test sequence ts_i:

$$IDS_i = FC_{i,1} \cap FC_{i,2} \cap \cdots \cap FC_{i,m} \ . \tag{15}$$

It will be formed by the intersection of conflict sets for every symptom, so each element in it can explain all the symptoms caused by the execution of ts_i.

5B) Removing the candidates which conflict with history data. A processing method will be done for each candidate in IDS.

1) For each diagnostic candidate $dc_{i,j}$ which has faulty transition with output fault, check the history test data for all outputs of this transition t_i. If for all found executions of t_i, their corresponding observed outputs are equal to output in this sequence ts_i execution, which means this faulty t_i can explain all observations, then this candidate is reserved. Otherwise, it will be deleted from the IDS.
2) For each diagnostic candidate $dc_{i,j}$ which has faulty transition with transfer fault, check the history test data for all ending states of t_i. If for all found executions of t_i, their corresponding ending states are equal to t_i in this sequence ts_i execution, which means this faulty t_i can explain all observations, then this candidate is reserved. Otherwise, it will be deleted from the IDS.

5C) After reducing the IDS, we get the final diagnostic candidate set "FDS".

Depending on the elements in the FDS, the following different actions might be chosen:

Case 1: If the FDS has single element, this candidate is the faulty implementation.

Case 2: If the FDS is not a singleton element set, we will describe the fault information using a faulty sub-sequence. The details will be discussed in Algorithm 2.

B. Test sequence generation

The test sequences are generated one by one from the DFSM model of software at test decision time. The DFSM model is finite state machine model with weight value and Boolean value for each transition. With the fault information get from the execution of test sequence, the DFSM model will be reconfigured after each test execution through updating the weight value and Boolean value.

A FSM model can be represented by a directed graph $G = (V, E)$. We use the weight value for each edge to describe the test cost of this transition execution. The initial value for each transition is $w_i = 1$. We suppose that there are m transitions in FSM model, and all the weight values compose a weight vector as one parameter of the FSM model,

$$W = [w_1, w_2, \cdots, w_m]. \tag{16}$$

Its value will be changed after one test execution. The change rule will be discussed with the Algorithm 1.

We define $\bullet$ as the transition concatenation operation, such as $t_1 \bullet t_2$ stands for transition t_2 is carried out after t_1. $t_1 \bullet t_2$ is denoted as a compound transition. Its value is computed through summing the weight value of every transition. E.g., the weight value for t_1, t_2 is assumed as w_1, w_2, such that the value of $t_1 \bullet t_2$ is $w_1 + w_2$.

At same time, we use a Boolean value to mark each transition have been tested or not. The value 0 is assigned to transition means that this transition has been tested. But 1 means it has not been tested. The Boolean value for all transitions compose another parameter vector of the FSM model,

$$B = [b_1, b_2, \cdots, b_m]. \tag{17}$$

Algorithm 1: The algorithm for test sequence generation is described as follows.

Step1: Generation of the initial FSM model for software.

We construct the FSM model for software to describe the behaviors of the software using the trace of events that record significant changes in the state of software.

Step2: Given the initial weight value and the Boolean value for each transition.

The initial test cost value vector is

$C = [c_1, c_2, \cdots, c_n] = [1, 1, \cdots, 1]$.

The initial Boolean value vector is

$B = [1, 1, \cdots, 1]$.

Then the FSM become a valued directed graph.

Step 3: We will do the following steps recursively.

Step 3.1: Generating test sequence at time *k*.

Generate the test sequence for the untested transition which is the nearest one to the initial state on the FSM graph (least cost from the initial state to the head state of the transition to be tested). The test sequence t_k is composed of the preamble sequence, the transition to be tested, and the postamble sequence. The expected outputs $o_1, o_2, \cdots, o_{m_i}$ of this sequence are generated from the FSM model.

The preamble sequence is

$(t_1, t_2, \cdots t_j)$.

$t_j(t_{j-1}(q_{j-2}, i_{j-1}), i_j)$ is the initial state of the transition to be tested. The weight value for each transition is w_k , the shortest preamble sequence is the sequence which has the least test cost, $\min(w_1 + w_2 + \cdots + w_j)$.

The UIO sequence for the tail state is generated according to the method in section II.

Step3.2: Executing the test sequence.

We execute the test sequence generated by step 3.1, and observe the outputs generated by each transition.

Step 3.3: Generating the diagnostic candidates

We follow the fault detection discussed in last section to do the step 3, 4, 5.

Step 3.4: Updating the model

Based on diagnostic candidates, the following three actions might be chosen:

Case 1: If no fault was detected, the Boolean value for transitions which have been verified through executing this test sequence is assigned 0. As stated in section II: if test sequence of transition i is included in test sequence j , the testing for the transition i can be performed by the test sequence j as well. So, more than one transition can be verified by one test execution,

$\{b_{t_1} = 0, b_{t_2} = 0, \cdots, b_{t_i} = 0\}$.

The test revenue for this execution is the transitions to be verified through this sequence which has 1 value before the execution of this test. Such as sequence t_2, t_4, t_7, t_{11} can verify transition t_2, t_4, t_7 . If the observed outputs after execution of this test is equal to the expected ones, such that $\{b_{t_2} = 0, b_{t_4} = 0, b_{t_7} = 0\}$. The test revenue of this test is $\{t_2, t_4, t_7\}$.

Case 2: If we can confirm the fault position, then the weight of corresponding transition is assigned infinitude, $w_i = \inf$, to demonstrate that this transition is fault and cannot be carried out rightly to verify other transitions. With the infinitude value, it will not be selected as part of test sequence. In practice, we usually assign a number large enough to implement this approach in computer, such as $w_i = 1000$.

Case 3: If the *FDS* contains more than one diagnostic candidates, we use the faulty sub-sequence to describe the fault information for next test decision making.

Definition 8: faulty sub-sequence. If we can be sure that the symptoms were resulted in by sequence of transitions

$t_{i,l}, t_{i,l+1} \cdots, t_{i,j}$,

which is part of the test sequence, we called it faulty sub-sequence.

Usually, we cannot get the confirm fault information after one test execution. In this paper we use the faulty sub-sequence to describe the fault information which is applied to decide following test sequence. The faulty sub-sequence contains the fault position of implementation, but not the exact position. The length of the sub-sequence is more short the fault isolation more accurate. The detailed method for getting shortest faulty sub-sequence is discussed in Algorithm 2. We assume that after this, the faulty sub-sequence is reduced as

$t_{i,l}, t_{i,l+1}, \cdots, t_{i,l+j}$.

The weight and Boolean values for each transition stay the same at this situation. But the weight value of this sub-sequence become infinitude, $w_l + \cdots + w_{l+j} = \inf$, to demonstrate that this sub-sequence has one fault at least and cannot be carried out rightly to verify other transitions. So in the following test, all test sequences cannot contain this sub-sequence as part of themselves.

Step 3.5: Updating the faulty sub-sequence set

Two operations will be done for the faulty sub-sequence set to get the newest ones:

1) Add the new faulty sub-sequence obtained from last step in it.

$FssS_{i+1} = FssS_i \bigcup \{Fss_i\}$.

2) Cut the history faulty sub-sequence in it

If this test execution successes, we can cut each element in the faulty sub-sequence set *FssS* according to the right transition information supplied by this test based on the two cutting rules. If new confirm faulty transition information can be get from the faulty sub-sequence set, we will update the weight value of transitions.

Step 3.6: Stopping dynamic test process.

If for all transitions, $\forall b_{t_i} = 0$ or for all executable test paths, $\forall r_{Ts_i} = 0$,the additional test might be no meaningful. So we can stop test generation.

Step 4: Additional test for diagnostic candidates.

With current observable events and controllable events, we may not get the unique diagnostic for software. At this situation, different points of observation and control might be needed in order to reduce the number of diagnostic candidates.

Now we will discuss the method of getting shortest faulty sub-sequence. FSM is a determinate machine, then we have:

Theorem 1: In two different executions, if the FSM will both arrive at state q_i , and the same input for this state is applied, such that the same end state will be arrived. At same time, the same output is observed when state is transferred to q_j , This is independent of the preamble sequence of state q_i .

Based on this, we have the following two cutting rules.

Rule 1: T is a faulty sub-sequence, and $|T| > 1$. If $T = t_1 \bullet T'$ and t_1 is a right transition, then T' is a shorter faulty sub-sequence with same faults which exist in T .

Proof. We suppose that t_e is the first faulty transition in T . If $e = 1$, then t_1 is the faulty implementation. This conflicts with the precondition t_1 is a right transition. So $t_e \in T'$, and T' is a faulty sub-sequence.

Rule 2: T is a faulty sub-sequence, and $|T|>1$. If $T = T'\bullet t_l$ and t_l is a right transition, then T' is a shorter faulty sub-sequence.

Proof. We suppose that t_e is the last faulty transition. If $e = l$, then t_l is the faulty implementation. This conflicts with the precondition t_l is a right transition. So $t_e \in T'$, and T' is a faulty sub-sequence.

Algorithm 2: In order to get the shortest faulty sub-sequence after one test execution fails, we do the following operations.

1) We first generate the initial faulty sub-sequence after the execution of certain test sequence

$t_{i,1}, t_{i,2}, \cdots, t_{i,m}$.

The set $\{o_{i,j} \neq \hat{o}_{i,j}, \cdots, o_{i,j+m} \neq \hat{o}_{i,j+m}\}$ is the symptom set generated by the execution of this sequence. $o_{i,j} \neq \hat{o}_{i,j}$ is the first symptom generated by transition $t_{i,j}$. Such that the sub-sequence

$t_{i,1}, t_{i,2}, \cdots, t_{i,j}$

has faulty transition implementation is the unique confirm information, when we cannot get the confirm fault information for the observation of this test. And it is the initial faulty sub-sequence.

2) Then we check the head and end transition of this sub-sequence by means of the two cutting rules based on history data. We will do this in the faulty sub-sequence until there are no head transitions or end transitions to be proved to be right implementation by history data.

From the previous discussion, we can see that in each test decision and test execution step, with the current cognition, a test sequence having the least test cost for certain transition is generated. With the execution of this test sequence, we have more information about the implementation of software and the model of software is updated dynamically. At the end of test, we might get the implementation model of software which has its faulty implementation information.

The test history is one optimal test sequence set:

$Ts_{opt} = \{ts_1, ts_2, \ldots, ts_m\}$.

This recursive process is formally defined as:

$$\begin{aligned}
&Model ::= (Ts, D, f_1, f_2, M);\\
&W_0 = [1,1,\cdots,1]; B_0 = [1,1,\cdots,1];\\
&FssS_0 = \varnothing; f_1^0 = 0; f_2^0 = 0;\\
&t^k = \min(|preamble(t)|); ts_{k+1} = T_s(t^k, M_k); \qquad (18)\\
&FssS_{k+1} = F_d(FssS_k, ts_{k+1}); W_{k+1} = W(W_k, ts_{k+1});\\
&B_{k+1} = B(B_k, ts_{k+1}); M_{k+1} = M(W_{k+1}, B_{k+1});\\
&f_1^{k+1} = f_1^k \cup r(ts_{k+1});\ f_2^{k+1} = f_2^k \cup c(ts_{k+1}).
\end{aligned}$$

Where, $\min(|preamble(t)|)$ is a function to compute the transition which is the nearest one to the initial sate; $T_s(t^k, M_k)$ is a function to compute the optimal test sequence for transition t^k at time k; $F_d(FssS_k, ts_{k+1})$ is a function to compute the faulty sub-sequence set after test execution for transition t^k; $W(W_k, ts_{k+1})$ is a function to compute the new weight value vector of FSM; $B(B_k, ts_{k+1})$ is a function to compute the new Boolean value vector of FSM; $M(W_{k+1}, B_{k+1})$ means updating the parameter of FSM.

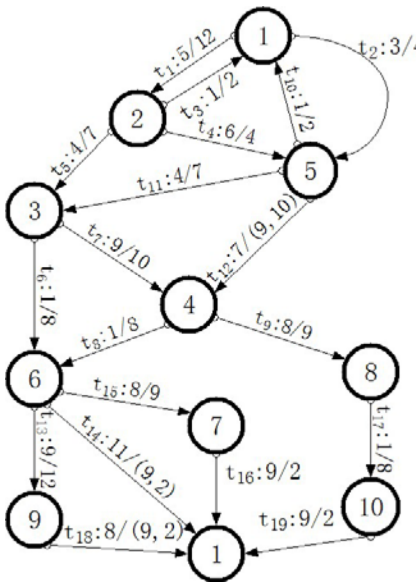

Figure 3. Simplified FSM for TCP

V. EXPERIMENT AND COMPARISONS

In this section, our method will be compared with the traditional test method with FSM (TFSM) and DCTM in terms of test cost, test coverage rate and the number of fake test results after applying them to transfer control protocol (TCP) test. The DCTM method applying for TCP has already been discussed in [11].

A. Finite state machine model of TCP

FSM of TCP that appeared in paper [18] is adopted for illustration. The simplified FSM contains a part of transmission control (i.e., connection setup and release) of TCP except data transmission (i.e., fragmentation and duplication) and the timer part. In case of a transition with no output, "null" is added for the output set. For simplicity, identifiers for states, inputs, and outputs are replaced with natural numbers. The state mapping and event mapping were generated in Tables III, respectively. Based on this, the TCP FSM is shown in Fig 3. There are

TABLE III.
STATE AND EVENT MAPPING OF TCP FSM

State	**Number**	**Event**	**Number**
Closed	1	close	1
Listen	2	closed	2
SYN_Revd	3	active_open	3
Estab	4	SYN	4
SYN_Sent	5	passive_open	5
FIN_Wait1	6	send_data	6
Closing	7	SYN_ACK	7
Close_Wait	8	FIN	8
FIN_Wait2	9	ACK	9
Last_ACK	10	established	10
		FIN_ACK	11
		null	12

TABLE IV.
UNIQUE INPUT/OUTPUT SEQUENCES FOR EACH STATE

State	UIO sequences
1	$t_1 : 5/12$
2	$t_3 : 1/2$
3	$t_6 : 1/8$
4	$t_9 : 8/9$
5	$t_{11} : 4/7$
6	$t_{13} : 9/12$
7	$t_{16} : 9/2$
8	$t_{17} : 1/8, t_{19} : 9/2$
9	$t_{18} : 8/(9,2)$
10	$t_{19} : 9/2, t_1 : 5/12$

a total of 10 states and 19 transitions in the TCP FSM.

With the traditional method shown in Section II, the UIO sequences for every state are shown in Table IV.

B. Simulation results and comparison

Test sequences generated with the U-method are shown in Table V.

Considering the fault types discussed in Section II.B, faults for a FSM can be classified into the following three cases [19]:

1) Produce an unexpected output for a given input and move to an expected state;
2) Produce an expected output for a given input and move to an unexpected state;
3) Produce an unexpected output for a given input and move to an unexpected state.

For simplicity of simulation, the faults given by case 2) and case 3) are considered in this paper, because the unexpected output is usually easy to be detected and it usually does not influence the corresponding observation of the other transitions.

Let us study one faulty transition situation and assume that t_6 is the unique faulty transition with transfer fault (the ending state of this transition is implemented as $\delta'(3,1) = 7$).

As discussed in Section II, when we use the traditional method to test the implementation, we will get the following test result: $t_1, t_2, t_3, t_4, t_5, t_7, t_8, t_9, t_{10}, t_{11}, t_{12}, t_{17}, t_{19}$ are assigned "pass" verdicts; $t_6, t_{13}, t_{14}, t_{15}, t_{16}, t_{18}$ are assigned "fail" verdicts. $t_{13}, t_{14}, t_{15}, t_{16}, t_{18}$ are fake results because of dependence between t_6 and them. At the same time, they were not covered actually.

With our method, for transition t_1, test sequence $ts_1 = t_1, t_3$ was generated based on M_0 and it passed test. Update the Boolean value of transition t_1 as 0. Then we generated the test sequences for t_2, t_3, t_4, t_5, they all passed tests. Then the Boolean vector of FSM was

$$B = [0,0,0,0,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1].$$

But when we applied $ts_6 = t_1, t_5, t_6, t_{13}$, we observed the conflict

TABLE V.
TEST SEQUENCES FOR TCP

Transition	Test sequences	Transition	Test sequences
1	t_1, t_3		
2	t_2, t_{11}	11	t_2, t_{11}, t_6
3	t_1, t_3, t_1	12	t_2, t_{12}, t_9
4	t_1, t_4, t_{11}	13	$t_1, t_5, t_6, t_{13}, t_{18}$
5	t_1, t_5, t_6	14	$t_1, t_5, t_6, t_{13}, t_{18}$
6	t_1, t_5, t_6, t_{13}	15	$t_1, t_5, t_6, t_{15}, t_{16}$
7	t_1, t_5, t_7, t_9	16	$t_1, t_5, t_6, t_{16}, t_1$
8	t_2, t_{12}, t_8, t_{13}	17	$t_2, t_{12}, t_9, t_{17}, t_{19}, t_1$
9	$t_2, t_{12}, t_9, t_{17}, t_{19}$	18	$t_1, t_5, t_6, t_{13}, t_{18}, t_1$
10	t_2, t_{10}, t_1	19	$t_2, t_{12}, t_9, t_{17}, t_{19}, t_1$

$$(o_{6,4} = 12) \neq (\hat{o}_{6,4} = 2).$$

With step 4 of the fault detection method, more than one diagnostic candidate can explain this observation, such as

$$DC_{6,1} : \lambda(6,9) = 2,\ DC_{6,2} : \delta(3,1) = 7$$

and etc. So we use the faulty sub-sequence to describe the fault information. The initial faulty sub-sequence was

$$Fss_1 = t_1, t_5, t_6, t_{13}.$$

With the cutting rules for faulty sub-sequence, because t_1, t_5 have been verified to be right, the shortest faulty sub-sequence was

$$Fss_1 = t_6, t_{13}.$$

Then the weight value for sub-path $t_6 \bullet t_{13}$ was assigned infinite, and the following test sequences should not contain this sub-path in them.

We generated the test sequences for $t_7, t_8, t_9, t_{10}, t_{11}, t_{12}$, and all of them pass tests. With the model M_{12}, the test sequence $ts_{13} = t_2, t_{12}, t_8, t_{13}, t_{18}$ for transition 13 was generated. This is different from the generation with traditional method $ts'_{13} = t_1, t_5, t_6, t_{13}, t_{18}$. The sub-path $t_6 \bullet t_{13}$ had infinite weight, so ts'_{13} is not least cost test sequence for t_{13}. ts_{13} passed test. With this result, the faulty sub-sequence $Fss_1 = t_6, t_{13}$ was cut to one transition t_6. With the execution data of ts_{11}, we can conclude that t_6 has right output. So t_6 has a transfer fault, two diagnostic candidates can explain the observation of ts_6,

$$\{FC_{6,1} : \delta(3,1) = 7, FC_{6,1} : \delta(3,1) = 10\}$$

Up to now, the unique faulty transition was confirmed,

TABLE VI.
TEST RESULTS WITH t_6 FAULT

	TFSM				DFSM				DCTM			
	C	P	F	W	C	P	F	U	C	P	F	W
t_6	72	13	6	5	72	18	1	0	98	18	1	0

Note: C stands for test cost which is measured by the weight value sum of transitions have been executed. P stands for the number of transitions pass tests. F stands for the number of transitions fail tests. W stands for the number of transitions with fake test results. U stands for the number of transitions which have not be tested.

TABLE VII.
TEST RESULTS WITH SINGLE FAULT

	TFSM				DFSM				DCTM			
	C	**P**	**F**	**W**	**C**	**P**	**F**	**U**	**C**	**P**	**F**	**W**
t_1	72	8	11	10	63	15	4	3	61	15	4	3
t_2	72	11	8	7	81	18	1	0	59	18	1	0
t_3	72	18	1	0	72	18	1	0	46	18	1	0
t_4	72	18	1	0	72	18	1	0	57	18	1	0
t_5	72	11	8	7	72	18	1	0	57	18	1	0
t_6	72	13	6	5	72	18	1	0	98	18	1	0
t_7	72	18	1	0	72	18	1	0	129	18	1	0
t_8	72	18	1	0	72	18	1	0	144	18	1	0
t_9	72	16	3	2	72	16	3	2	92	16	3	2
t_{10}	72	18	1	0	72	18	1	0	54	18	1	0
t_{11}	72	18	1	0	72	18	1	0	81	18	1	0
t_{12}	72	14	5	4	84	18	1	0	85	18	1	0
t_{13}	72	17	2	1	66	17	2	1	64	17	2	1
t_{14}	72	18	1	0	72	18	1	0	65	18	1	0
t_{15}	72	17	2	1	66	17	2	1	64	17	2	1
t_{16}	72	18	1	0	72	18	1	0	101	18	1	0
t_{17}	72	16	3	2	66	16	3	2	54	16	3	2
t_{18}	72	18	1	0	72	18	1	0	101	18	1	0
t_{19}	72	16	3	2	66	16	3	2	54	16	3	2
E	72	15.9	3.1	2.1	70.9	17.4	1.6	0.6	77.2	17.4	1.6	0.6
I	/	18	1	0	/	18	1	0	/	18	1	0

Note: E stands for the average value of each line. I stands for the ideal value of each line. The faulty implementation is assumed as follows:

$t_1:\delta'(1,5)=5; t_2:\delta'(1,3)=2; t_3:\delta'(2,1)=4; t_4:\delta'(2,6)=4;$

$t_5:\delta'(2,4)=4; t_6:\delta'(3,1)=7; t_7:\delta'(3,9)=5; t_8:\delta'(4,1)=7;$

$t_9:\delta'(4,8)=7; t_{10}:\delta'(5,1)=4; t_{11}:\delta'(5,4)=4; t_{12}:\delta'(5,7)=3;$

$t_{13}:\delta'(6,9)=1; t_{14}:\delta'(6,11)=9; t_{15}:\delta'(6,8)=1; t_{16}:\delta'(7,9)=10;$

$t_{17}:\delta'(8,1)=4; t_{18}:\delta'(9,8)=7; t_{19}:\delta'(10,9)=7.$

so the weight value for transition t_6 was assigned infinite. The character vectors for FSM were

$$C=[1,1,1,1,1,\text{inf},1,1,1,1,1,1,1,1,1,1,1,1,1],$$

$$B=[0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,1].$$

Then we generated test sequences for the remaining transitions, and they all passed tests. The test history is:

$ts_1=t_1,t_3; ts_2=t_2,t_{11}; ts_3=t_1,t_3,t_1; ts_4=t_1,t_4,t_{11}; ts_5=t_1,t_5,t_6;$

$ts_6=t_1,t_5,t_6,t_{13}; ts_7=t_1,t_5,t_7,t_9; ts_8=t_2,t_{12},t_8,t_{13};$

$ts_9=t_2,t_{12},t_9,t_{17},t_{19}; ts_{10}=t_2,t_{10},t_{11}; ts_{11}=t_2,t_{11},t_6;$

$ts_{12}=t_2,t_{12},t_9; ts_{13}=t_2,t_{12},t_8,t_{13},t_{18}; ts_{14}=t_2,t_{12},t_8,t_{14},t_1;$

$ts_{15}=t_2,t_{12},t_8,t_{15},t_{16}; ts_{16}=t_2,t_{12},t_8,t_{15},t_{16},t_1;$

$ts_{17}=t_2,t_{12},t_9,t_{17},t_{19},t_1; ts_{18}=t_2,t_{12},t_8,t_{13},t_{18},t_1;$

$ts_{19}=t_2,t_{12},t_9,t_{17},t_{19},t_1$

This is one optimal test sequence set for software implementation with fault $\delta'(3,1)=7$. The comparison with traditional method is shown in Table VI.

From the test results shown in Table VI, we can see that our DFSM method has higher test coverage rate and expends same test cost compared with the TFSM method. In t_6 fault situation, the test coverage rate is closer to the ideal with DFSM method. The same test coverage rate is obtained by DCTM, but its test cost is higher than DFSM. We study all single fault situations, and the simulation results are shown in Table VII.

Considering one fault situation, the number of "pass" transitions is 17.4 by the method DFSM or DCTM. But only 15.9 transitions pass when we use TFSM method. We will compare DFSM with DCTM and TFSM from three aspects: test coverage rate, effective test efficiency and number of fake test results.

Test coverage rate: From the simulation experiments, in average view, DFSM has same test coverage rate with DCTM but with less test cost. They both have higher test coverage rate than TFSM. The test coverage rate with DCTM and DFSM are closer to the ideal. The test coverage rate is computed by

$$t_c / t_a,$$

where, t_c means the number of covered transitions; t_a means the number of all transitions. With the simulation results, the test coverage rate in each fault situation is figured in Fig. 4. The lines for DFSM and DCTM are superposition in the figure, because they have same test coverage rate.

Effective test efficiency: We use the following equation to compute the effective test efficiency:

$$t_c / TC,$$

where, TC means test cost and t_c stands for transitions which are covered. The average effective test efficiency of the three methods are:

$$TFSM_a = t_c / TC_{all} = 321/1368 = 0.235$$

$$DFSM_a = t_c / TC_{all} = 350/1356 = 0.258.$$

$$TFSM_a = t_c / TC_{all} = 350/1466 = 0.239$$

We can see that DFSM has highest average effective test efficiency, in average view. Compared with DCTM, DFSM has less redundant tests, because DCTM detect

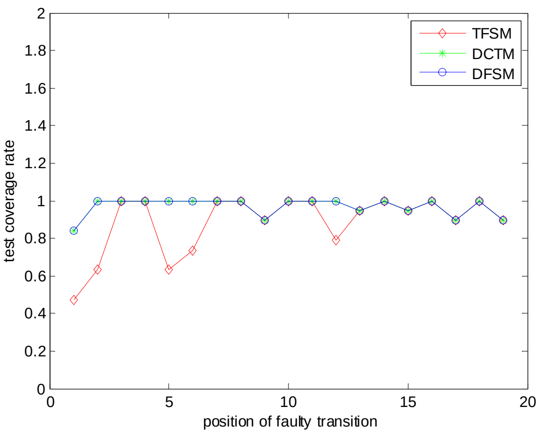

Figure 4. Test coverage rate

faults based on all executable paths for one transition fail. But DFSM reasons with history data to detect fault and uses faulty sub-sequence to describe fault information. The effective test efficiency in each fault situation for every method is shown in Fig. 5.

Number of fake test results: With DFSM and DCTM methods, the fake test results are much less than TFSM. The DFSM method uses U to denote the transitions which have not been tested. In future testing activity, we can employ more sensors to test these transitions to get the confirm fault information. But with DTCM method, when the executable test sequences for certain transition are all wrong, the "fail" verdict is assigned to this transition. This may cause fake test results also, such as t_{17} in t_9 fault situation. Non preamble sequence can lead the state8 of FSM, so t_{17} had not been tested from beginning to end. It is inappropriate to assign a "fail" verdict to it. The number of fake test results is shown in Fig. 6 for every method. The number in this figure for DFSM is the number of transitions which have not been tested which is same with the fake test result number caused by DTCM method.

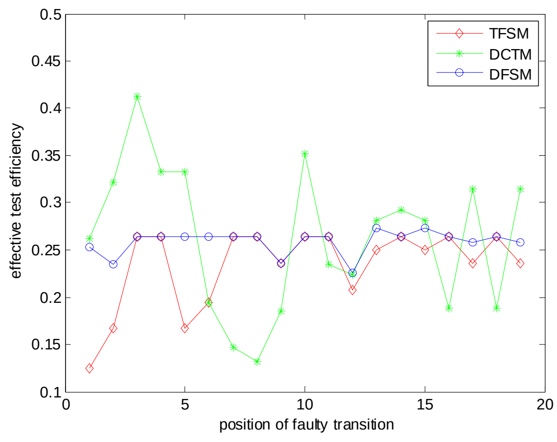

Figure 5. Effective test efficiency

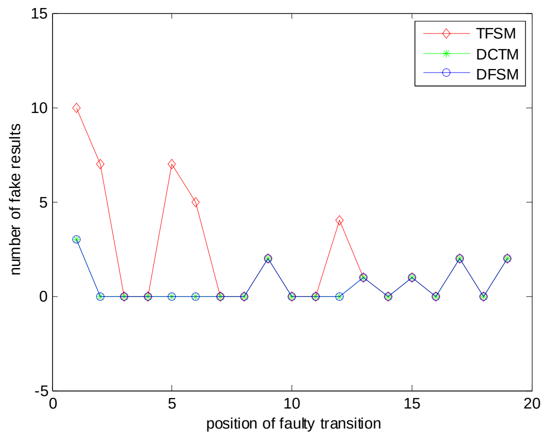

Figure 6. Number of fake test results

VI. Conclusion

Traditional FSM based test methods carry out tests based on fix test sequence set and the test execution process is independent with test analysis process. This causes the problems shown in section II. In this paper, considering the fault effects on testing and the dependences between transitions, we propose a stochastic combinatorial optimization model to describe the test sequence generation problem. At same time, a recursive algorithm is proposed to give one optimal solution for the test sequence generation. In our method, the software test execution process and fault detection process compose a close-loop process. Moreover, a weighted FSM is modeled for the software under test, and the weight value and Boolean value of each transition may be changed after the test execution and fault detection. We call this method dynamic finite state machine (DFSM). At each time a test sequence is generated according to the current weighted FSM model.

The simulation results in Section V show that the DFSM method is an efficient approach to solve the three problems. It has higher test coverage rate and more effective test efficiency than traditional FSM based test methods. Specially, the fake test results are much reduced. Compared with related work DCTM, the DFSM method has less test cost in average view. At same time, the fault assumption and detection method are more close to real application. The DCTM only needs to maintain a weighted finite state machine but doesn't need to generate all possible preamble and postamble sequences for every transition which is the way adopted by DCTM. So it is easy to be implemented.

In short, the DFSM method provides a convenient way for testing with finite state machine.

Acknowledgment

This work was supported in part by the National Key Technology R&D Program under Grant 2009BAG12A08, the R&D Foundation of the Ministry of Railways under Grant 2009X003 and the Research Foundation of Beijing National Railway Research and Design Institute of Signal and Communication.

References

[1] T. S. Chow, "Testing software design modeled by finite-state machines," IEEE Trans. Software Eng., vol. SE-4, Issue 3, pp. 178-187, 1978.

[2] G. Gonenc, "A method for the design of fault detection experiments," IEEE Trans. Computers, vol. 19, issue 6, pp. 551-558, June 1970.

[3] A. Petrenko and N. Yevtushenko, "Test suite generation from a FSM with a given type of implementation errors,'' Proc. IFIP 12th Int. Symp. Protocol specification. Testing, and Verification XII, North-Holland, pp. 229-243, 1992.

[4] D. P. Sidhu and T. K. Leung, "Formal methods for protocol testing: A detailed study," IEEE Trans. Software Eng., vol. 15, Issue 4, pp. 413-426, Apr. 1989.

[5] M. P. Vasilevskii, "Failure diagnosis of automata," Cybernetics, Vol.9, No. 4, pp. 653-665, 1973.

[6] Gregor v. Bochmann and Alexamdre Petrenko, “Protocol testing: review of methods and relevance for software testing,” Proceedings of the 1994 ACM SIGSOFT international symposium on Software testing and analysis, pp.109-124, 1994.
[7] Yanghee Choi, Dongkyun Kim, Jaecheol Kim, and et al., “Protocol test sequence generation using UIO and BUIO,” 1995 IEEE International Conference on communications, pp. 362-366, 1995.
[8] S. Fujiwara, G von Bochmann, F.Khendek and et al., “Test selection based finite state models,” IEEE Transactions On Software Engineer, Vol.17, Issue 6, pp.591-603, 1991.
[9] S. C. Pinto Ferraz Fabbri, M. E. Delamaro, J. C. Maldonado and et al., “Mutation analysis testing for finite state machine,” Proceedings of 5th International Symposium on Software Reliability Engineer, pp.220-229, 1994.
[10] Chanson, S.T. and Li, Q., “On static and dynamic test case selections in protocol conformance testing,” In: The 5th Int’l Workshop on Protocol Test Systems, pp.225-267, 1992.
[11] Myungchul Kim, Sangjo Yoo, and et al., “A dynamic protocol conformance test method,” The Journal of Systems and Software, 67, pp.31-43, 2003.
[12] John E. Hopcroft, Rajeev Motwani, Jeffrey D. Ullman. “Introduction to automata theory, languages, and computation (2nd Edition),” Pearson Education. ISBN 0-201-44124-1, 2000.
[13] Krishan Sabnani and Anton Dahbura, ‘‘A protocol test generation procedure,’’ Computer Networks and ISDN Systems, vol. 15, no. 4, pp. 285-297, 1988.
[14] G. v. Bochmann, A. Das, R. Dossouli, M. Dubuc and et al., “Fault model in testing”, Proceedings of the IFIP TC6/WG6.1 Fourth International Workshop on Protocol Test Systems IV, pp.17-30, October 15-17, 1991.
[15] Ghedamsi A., and Bochmann G. Von, “Test result analysis and diagnostics for finite state machines,” Proceeding of the 12th international Conference on Distributed Computing Systems, pp.244-251, 1992.
[16] Miller R E. and Arisha K A., “Fault identification in networks by passive testing,” Proceeding of the 34th Annual Simulation Symposium, pp.277-284, 2001.
[17] Miller R E. and Arisha K A. “On fault location in networks by passive testing,” Proceeding of the IEEE international Performance, Computing, and Communications Conference, pp.281-287, 2000.
[18] Seol, S., Kim, M., Kang, S., Park, Y., and Choe, Y., “Interoperability test suite derivation for the TCP,” IFIP Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE XII) and Protocol Specification, Testing and Verification (PSTV XIX), pp.357-376, 1999.
[19] Sidhu, D.P., Leung, T.K., “Formal methods for protocol testing: a detailed study,” IEEE Trans. Software Eng., Vol. 15, Issue 4, pp.413-426. 1989.

Shuai Wang was born in Changchun, Jilin Province, China, on April 3, 1981. He received his B.S. degree in control science and engineering from Beijing Institute of Technology University, Beijing, China in 2004. Currently, he is a direct PH.D research candidate with control science and engineering at Tsinghua University since 2004. He major in system test, fault diagnosis and reliability analysis.

Yindong Ji was born in Beijing, China in 1962. He received his B.S. and M.S. all from the Department of Automation, at Tsinghua University, in 1985 and 1989, respectively. His main research areas are digital signal process, fault diagnosis, modeling & simulations. He is a member of IEEE.

Prof. JI is with the Department of Automation, and Tsinghua National Laboratory for Information Science and Technology, Tsinghua University, Beijing, China. He has published over 60 papers in journals. His current research interest is in the area of train control system of high speed railway.

Shiyuan Yang was born in Shanghai, China, in 1945. He received his B.S. and M.S degree from Tsinghua University in 1970 and 1981, respectively.

Currently, he is a Professor in automation of department in Tsinghua University. He is an Associate Director of the FTC committee, China. His main research interests are home automation network, test technology, electronic technology application, system fault diagnosing.