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 Ji¹², Shiyuan Yang¹
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 preamble 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 faulty transitions in the preamble sequence or the postamble sequence may interfere with the judgment 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 cannot be satisfied 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 subsequence 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) $\Sigma$ is the set of finite events which contains both the input events and output events, $\Sigma = I \cup O$;
3) $\delta: Q \times \Sigma \rightarrow Q$ are the state transition functions;
4) $q_0$ is the initial state;
5) $\lambda: Q \times \Sigma \rightarrow \Sigma$ 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 $V = \{v_1, \ldots, v_n\}$ 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 $L = a_i / o_i$, $L^{(i)} = a_j$ and $L^{(o)} = o_i$.

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 $a_k / o_k$ takes place in three steps.

1) The implementation is loaded into state $q_i$
2) Input $a_k$ is applied and the output is checked to verify that whether it is $a_j$, 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.
State  | U/O sequence | State  | U/O sequence  
|-------|-------------|-------|-------------|
| \(q_1\) | \(b/x, b/y, z\) | \(q_1\) | \(a/x, c/z\) 
| \(q_2\) | \(b/y\) | \(q_2\) | \(a/z\) 
| \(q_3\) | \(c/y, a/z\) | 

\(M_{\text{prev}}\), we say that \(I\) has no fault with respect to \(M\); and if \(M \neq M_{\text{prev}}\), we define the faulty types as follows:

1) Output fault: We say that a transition in \(I\) has an output fault if there exists \(M'\) such that \(M' = M_{\text{prev}}\) and \(M'\) can be obtained from \(M\) by changing the output of certain transition.

\[ q_i = q_j \land \delta(q_i, a) = \delta'(q_j', a) \land \lambda(q_i, a) \neq \lambda'(q_j', a) \]

2) Transfer fault: We say that \(I\) has a transfer fault if there exists \(M'\), such that \(M' = M_{\text{next}}\) and \(M'\) can be obtained from \(M\) by changing the ending state of certain transition.

\[ q_i = q_j' \land \delta(q_i, a) \neq \delta'(q_j', a) \land \lambda(q_i, a) = \lambda'(q_j', a) \]

Based on these two faults, for a complete defined FSM, there exist \((n-1)(q-1)/2\) faulty situations. This is a 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 U/O sequence for each state is shown in Table I.

The generated sequences are in Table II.

Table I.

<table>
<thead>
<tr>
<th>State</th>
<th>U/O sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>(q_1)</td>
<td>(b/x, b/y, z)</td>
</tr>
<tr>
<td>(q_2)</td>
<td>(b/y)</td>
</tr>
<tr>
<td>(q_3)</td>
<td>(c/y, a/z)</td>
</tr>
</tbody>
</table>

Table II.

<table>
<thead>
<tr>
<th>Test aim</th>
<th>Input</th>
<th>Expected output</th>
<th>Transition sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>(b_1)</td>
<td>(l_1:q_0 \to q_0)</td>
<td>(r,a,b,h)</td>
<td>(x,x,y)</td>
</tr>
<tr>
<td>(b_2)</td>
<td>(l_2:q_0 \to q_1)</td>
<td>(r,b,b)</td>
<td>(x,y)</td>
</tr>
<tr>
<td>(b_3)</td>
<td>(l_3:q_0 \to q_1)</td>
<td>(r,c,a,c)</td>
<td>(y,x,z)</td>
</tr>
<tr>
<td>(b_4)</td>
<td>(l_4:q_1 \to q_2)</td>
<td>(r,b,h,c,a)</td>
<td>(x,y,y,z)</td>
</tr>
<tr>
<td>(b_5)</td>
<td>(l_5:q_1 \to q_2)</td>
<td>(r,b,h,a,a)</td>
<td>(x,y,z)</td>
</tr>
<tr>
<td>(b_6)</td>
<td>(l_6:q_2 \to q_4)</td>
<td>(r,b,h,b,a)</td>
<td>(x,y,y,z)</td>
</tr>
<tr>
<td>(b_7)</td>
<td>(l_7:q_2 \to q_4)</td>
<td>(r,b,h,c,a)</td>
<td>(y,x,y,z)</td>
</tr>
<tr>
<td>(b_8)</td>
<td>(l_8:q_1 \to q_2)</td>
<td>(r,c,b,c,a)</td>
<td>(y,x,y,z)</td>
</tr>
<tr>
<td>(b_9)</td>
<td>(l_9:q_3 \to q_4)</td>
<td>(r,c,a,a)</td>
<td>(y,x,z)</td>
</tr>
<tr>
<td>(b_{10})</td>
<td>(l_{10}:q_4 \to q_0)</td>
<td>(r,c,a,c)</td>
<td>(y,x,z)</td>
</tr>
<tr>
<td>(b_{11})</td>
<td>(l_{11}:q_4 \to q_0)</td>
<td>(r,c,a,a)</td>
<td>(y,x,z)</td>
</tr>
</tbody>
</table>
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. Based on this idea, we construct the stochastic combinatorial optimization model for test sequence optimization in this section.

### 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. Based 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 $t = k - 1$. $T = \{0, 1, 2, \ldots\}$ 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 $T_S$, $T_S = \{t_{S_1}, t_{S_2}, \ldots, t_{S_n}\}$

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

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

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

$T_i = \{t_j \cdots t_m, i \in [1, m], j \in [1, m], i \neq j\}$.                  (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.

$D_i = \{t_{S_j}, t_{S_j}, \cdots, t_{S_j}\}$.                              (4)

All executable test sequences compose the decision space.

$D = \{T_{S_1}, T_{S_2}, \ldots, T_{S_n}\}$.                          (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.
$R_r = r(d_r), i \in [1, n]$.  

In this paper, the value of $R_r$ is the transition which have been tested through the executing of $d_r$.

$$R_r = r(d_r) = \{t_i, t_{2i}, \ldots, t_{n_i}\} \cap T_i, j \in [1, n].$$

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

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

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 $d_i$.

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

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.

$$f_1 = \bigcup_{i=1}^n R_i = \bigcup_{i=1}^n r(d_i);$$

$$f_2 = \sum_{i=1}^n C_i = \sum_{i=1}^n c(d_i).$$

$r$ is the test ending time.

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

$$D = (d_1, d_2, \ldots, d_l).$$

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

$$Ts = (t_{s_1}, t_{s_2}, \ldots, t_{s_l}).$$

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

$$(f_1 = 0_0) \land \min(f_2).$$

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

$$Model ::= (T, D, f_1, f_2).$$

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:

$$R'_r = r(d_r) = a(f) r(d_r),$$

$$C'_i = c(d_i) = b(f) c(d_i),$$

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

Take the test sequence $ts_4$ for example. With traditional method, $R = r(ts_4) = \{t_2, t_{12}, t_{1}\}, C = c(ts_4) = 4$. Using our method, if $t_2$ is wrong, $R' = \emptyset$; if $t_{12}$ is wrong, $R' = \emptyset$; if $t_1$ is wrong, $R' = \{t_1\}$; if $t_{12}$ is wrong, $R' = \{t_{12}\}$.

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

$$Model ::= (T, D, f'_1, f'_2, F);$$

$$f'_1 = \bigcup_{i=1}^n R'_i = \bigcup_{i=1}^n a_i(f) r(d_i) = \bigcup_{i=1}^n a_i(f)(t_{i1}, t_{i2}, \ldots, t_{in_i}) \cap T_i$$

$$f'_2 = \sum_{i=1}^n C'_i = \sum_{i=1}^n b_i(f) c(d_i);$$

$$\max(f'_1); \min(f'_2).$$

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

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 $ts$ consists of $m$ transitions $t_1, t_2, \ldots, t_m$. Its corresponding sequence of expected outputs is written as $o_1, o_2, \ldots, o_m$, where $o_i$ is expected after transition $t_i$ happening.
Step 2: Generation of the observed outputs
After applying test sequence to the implementation of software, a corresponding observed sequence is written as $o_1, o_2, \ldots, o_n$ for each test sequence $t_s$.

Step 3: 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 $s_{o,j}$. The symptom set for test sequence $t_s$ execution is

$$S_t = \{s_{1,j}, s_{2,j}, \ldots, s_{m,j}\}.$$  

Step 4: Generation of the conflict set
For each symptom $s_{i,j}$, construct its corresponding conflict set

$$FC_{i,j} = \{f_{j,1}, f_{j,2}, \ldots, f_{j,m}\},$$

where, $f_{j,m}$ 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.

Step 5: Generation of the diagnostic candidates
Diagnostic candidates are components which are suspected to be faulty to explain the observation of test execution $t_s$. This is achieved by three steps:

5A) Computing the initial diagnostic candidate set “IDS” of test sequence $t_s$:

$$IDS_t = FC_{1,j} \cap FC_{2,j} \cap \cdots \cap FC_{m,j}.$$  

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 $t_s$.

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 $t_j$. If for all found executions of $t_j$, their corresponding observed outputs are equal to output in this sequence $t_s$ execution, which means this faulty $t_j$ 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 $t_j$. If for all found executions of $t_j$, their corresponding ending states are equal to $t_j$ in this sequence $t_s$, execution, which means this faulty $t_j$ 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-sequencce. 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 $w_t=1$. 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,

$$W = [w_1, w_2, \ldots, w_m].$$

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 $t_1$ is carried out after $t_2$. $t_1 \bullet t_2$ is denoted as a compound transition. Its value is computed through summing the weight value of every transition. E.g., the weight value for $t_1 \bullet t_2$ 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,

$$B = [b_1, b_2, \ldots, b_m].$$

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

Step 1: 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.

Step 2: Given the initial weight value and the Boolean value for each transition.

The initial test cost value vector is $c = [c_1, c_2, \ldots, c_n] = [1, 1, \cdots, 1]$. 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 graph).
the transition to be tested). The test sequence \( t_i \) is composed of the preamble sequence, the transition to be tested, and the postamble sequence. The expected outputs \( o_1,o_2,\ldots,o_m \) of this sequence are generated from the FSM model.

The preamble sequence is
\[
(t_1, t_2, \ldots , t_i).
\]

\( t_i \) is the initial state of the transition to be tested. The weight value for each transition is \( w_j \), the shortest preamble sequence is the sequence which has the least test cost, \( \min(w_1 + w_2 + \cdots + w_i) \).

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

**Step 3.2: 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, 
\[
\{b_0 = 0, b_1 = 0, \ldots , b_i = 0\}.
\]

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 \( t_2, t_4, t_7, t_9 \) 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 \( \{b_0 = 0, b_1 = 0, b_2 = 0\} \). The test revenue of this test is \( \{t_1, t_2, t_4\} \).

**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 
\[
l_{j,1}t_{j,1}l_{j,1}\cdots t_{j,j},
\]

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 
\[
l_{j,1}t_{j,1}\cdots t_{j,j}.
\]

The weight and Boolean values for each transition stay the same at this situation. But the weight value of this sub-sequence become infinitude, \( w_i + \cdots + w_{j,k} = \inf \), 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.
\[
F_{ssS_{i+1}} = F_{ssS_i} \cup \{F_{ssS_i}\}
\]

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 \( F_{ssS} \) 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, \( \forall b_i = 0 \) or for all executable test paths, \( \forall e_i = 0 \), 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 \( q_j \), 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 \( q_j \).

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

**Rule 1:** \( T \) is a faulty sub-sequence, and \( |T| > 1 \). If \( T = t_i \bullet T' \) and \( t_i \) 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 \( t_i \) is the faulty implementation. This conflicts with the precondition \( t_i \) 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 \( T = T \cdot t \) and \( t \) is a right transition, then \( T' \) is a shorter faulty sub-sequence.

Proof. We suppose that \( t \) is the last faulty transition. If \( e = l \), then \( t \) is the faulty implementation. This conflicts with the precondition \( t \) is a right transition. So \( t \in T' \), 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

\( t_{i,j}, t_{i,k}, \ldots, t_{i,m} \).

The set \( \{o_{i,j} \neq \hat{o}_{i,j}, \ldots, o_{i,j,m} \neq \hat{o}_{i,j,m}\} \) 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

\( t_{i,j}, t_{i,k}, \ldots, t_{i,j} \)

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:

\( T_{opt} = \{t_{s,1}, t_{s,2}, \ldots, t_{s,n}\} \).

This recursive process is formally defined as:

\[
Model := (T_{s}, D_{s}, j_{f}, f_{s}, M_{s});
W_{s} = [1,1,\ldots,1]; B_{s} = [1,1,\ldots,1];
FSS_{s} = \emptyset; f_{s}^{0} = 0; f_{s}^{n} = 0;
\]

\[
t^{k} = \min(\{\text{preamble}(i)\}; t_{s,i} = T_{s}(t^{k}, M_{s}));
\]

\[
FSS_{s+1} = F_{s}(FSS_{s}, t_{s,i}); W_{s+1} = W(W_{s}, t_{s,i});
B_{s+1} = B_{s}(t_{s,i}, t_{s,i}); M_{s+1} = M(W_{s}, B_{s}, t_{s,i});
\]

\*

\[
f_{s}^{i+1} = f_{s}^{i} \cup \text{r}(t_{s,i}), f_{s}^{i+1} = f_{s}^{k} \cup c(t_{s,i}).
\]

Where, \( \min(\{\text{preamble}(i)\}) \) is a function to compute the transition which is the nearest one to the initial state; \( T_{s}(t^{k}, M_{s}) \) is a function to compute the optimal test sequence for transition \( t^{k} \) at time \( k \); \( F_{s}(FSS_{s}, t_{s,i}) \) is a function to compute the faulty sub-sequence set after test execution for transition \( t^{k} \); \( W(W_{s}, t_{s,i}) \) is a function to compute the new weight value vector of FSM; \( B_{s}(t_{s,i}) \)

is a function to compute the new Boolean value vector of FSM; \( M(W_{s}, B_{s}, t_{s,i}) \) means updating the parameter of FSM.

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>
<thead>
<tr>
<th>Table III. STATE AND EVENT MAPPING OF TCP FSM</th>
</tr>
</thead>
<tbody>
<tr>
<td>State</td>
</tr>
<tr>
<td>-------------</td>
</tr>
<tr>
<td>Closed</td>
</tr>
<tr>
<td>Listen</td>
</tr>
<tr>
<td>SYN_Revd</td>
</tr>
<tr>
<td>Estab</td>
</tr>
<tr>
<td>SYN_Sent</td>
</tr>
<tr>
<td>FIN_Wait1</td>
</tr>
<tr>
<td>Closing</td>
</tr>
<tr>
<td>Close_Wait</td>
</tr>
<tr>
<td>FIN_Wait2</td>
</tr>
<tr>
<td>Last_ACK</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
The weight value for sub-path 6,3 is the unique faulty transition with transfer fault (the ending state of this transition is implemented as δ(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: \( t_1, t_2, t_1, t_3, t_5, t_7, t_5, t_6, t_8, t_9, t_10, t_1, t_12, t_13, t_10 \) 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 \( t_5, t_5, t_1 \) 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 \( B = [0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1] \).

But when we applied \( t_6, t_5, t_4, t_3 \), we observed the conflict \( (o_{t_6} = 12) \neq (o_{t_6} = 2) \).

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 \( FS_{t_6} = t_1, t_2, t_3, t_13 \).

With the cutting rules for faulty sub-sequence, because \( t_1, t_2 \) have been verified to be right, the shortest faulty sub-sequence was \( FS_{t_6} = t_6 \).

Then the weight value for sub-path \( t_6 \) was assigned infinite, and the following test sequences should not contain this sub-path in them.

We generated the test sequences for \( t_1, t_2, t_3, t_4, t_5, t_{10}, t_{11}, t_{12}, t_{13} \), and all of them pass tests. With the model \( M_{12} \), the test sequence \( t_{13} = t_2, t_3, t_4, t_13, t_{18} \) for transition 13 was generated. This is different from the generation with traditional method \( t_{13} = t_1, t_3, t_4, t_{13}, t_{18} \). The sub-path \( t_6 \) had infinite weight, so \( t_{13} \) is not least cost test sequence for \( t_{13} \). \( t_{13} \) passed test. With this result, the faulty sub-sequence \( FS_{t_6} = t_6 \) was cut to one transition \( t_6 \).

With the execution data of \( t_{13} \), we can conclude that \( t_6 \) has right output. So \( t_6 \) has a transfer fault, two diagnostic candidates can explain the observation of \( t_{6} \),

\[
\{ FC_{6,1} : \delta (3,1) = 7, FC_{6,2} : \delta (3,1) = 10 \}
\]

Up to now, the unique faulty transition was confirmed.

### Table IV. Unique Input/Output sequences for each state

<table>
<thead>
<tr>
<th>State</th>
<th>U/O sequences</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( t_1 : 5/12 )</td>
</tr>
<tr>
<td>2</td>
<td>( t_2 : 1/2 )</td>
</tr>
<tr>
<td>3</td>
<td>( t_3 : 1/8 )</td>
</tr>
<tr>
<td>4</td>
<td>( t_4 : 8/9 )</td>
</tr>
<tr>
<td>5</td>
<td>( t_5 : 4/7 )</td>
</tr>
<tr>
<td>6</td>
<td>( t_6 : 9/12 )</td>
</tr>
<tr>
<td>7</td>
<td>( t_7 : 9/2 )</td>
</tr>
<tr>
<td>8</td>
<td>( t_8 : 1/8, t_9 : 9/2 )</td>
</tr>
<tr>
<td>9</td>
<td>( t_{10} : 8/(9,2) )</td>
</tr>
<tr>
<td>10</td>
<td>( t_{10} : 9/2, t_1 : 5/12 )</td>
</tr>
</tbody>
</table>

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

With the traditional method shown in Section II, the U/O 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 \( \delta' (3,1) = 7 \)).

### Table V. Test sequences for TCP

<table>
<thead>
<tr>
<th>Transition</th>
<th>Test sequences</th>
<th>Transition</th>
<th>Test sequences</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( t_1, t_3 )</td>
<td>11</td>
<td>( t_2, t_3, t_6 )</td>
</tr>
<tr>
<td>2</td>
<td>( t_2, t_1 )</td>
<td>12</td>
<td>( t_2, t_3, t_6 )</td>
</tr>
<tr>
<td>3</td>
<td>( t_1, t_3, t_5 )</td>
<td>13</td>
<td>( t_1, t_3, t_4, t_5, t_3, t_8 )</td>
</tr>
<tr>
<td>4</td>
<td>( t_1, t_4, t_11 )</td>
<td>14</td>
<td>( t_1, t_3, t_4, t_5, t_3, t_8 )</td>
</tr>
<tr>
<td>5</td>
<td>( t_1, t_5, t_6 )</td>
<td>15</td>
<td>( t_1, t_3, t_4, t_5, t_3, t_8 )</td>
</tr>
<tr>
<td>6</td>
<td>( t_2, t_5, t_6, t_3 )</td>
<td>16</td>
<td>( t_1, t_3, t_4, t_5, t_3, t_8 )</td>
</tr>
<tr>
<td>7</td>
<td>( t_1, t_5, t_6, t_5 )</td>
<td>17</td>
<td>( t_1, t_3, t_4, t_5, t_3, t_8 )</td>
</tr>
<tr>
<td>8</td>
<td>( t_2, t_3, t_5, t_3, t_8 )</td>
<td>18</td>
<td>( t_1, t_3, t_4, t_5, t_3, t_8 )</td>
</tr>
<tr>
<td>9</td>
<td>( t_2, t_3, t_5, t_3, t_8 )</td>
<td>19</td>
<td>( t_1, t_3, t_4, t_5, t_3, t_8 )</td>
</tr>
</tbody>
</table>

Up to now, the unique faulty transition was confirmed.

### Table VI. Test results with \( t_6 \) fault

<table>
<thead>
<tr>
<th>TFSM</th>
<th>DFSM</th>
<th>DCTM</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>P</td>
<td>F</td>
</tr>
<tr>
<td>72</td>
<td>13</td>
<td>6</td>
</tr>
<tr>
<td>C</td>
<td>P</td>
<td>F</td>
</tr>
<tr>
<td>72</td>
<td>18</td>
<td>1</td>
</tr>
<tr>
<td>U</td>
<td>F</td>
<td>P</td>
</tr>
<tr>
<td>98</td>
<td>18</td>
<td>1</td>
</tr>
</tbody>
</table>

Note: C stands for test cost which is measured by the weight value sum of transitions has 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 without test results. U stands for the number of transitions which have not be tested.
The faulty implementation is assumed as follows:

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:

$$E = \frac{1}{n} \sum_{i=1}^{n} E_i$$

$$I = \frac{1}{n} \sum_{i=1}^{n} I_i$$

The character vectors for FSM were

$$C = \{1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1\}$$

$$B = \{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0\}$$

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

The test coverage rate is computed by

$$\frac{c_t}{t_c} \times 100\%$$

where, $c_t$ means the number of covered transitions; $t_c$ means the number of all transitions. With the simulation results, the test coverage rate in each fault situation is showed in Table VII.

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

$$\frac{c_t}{t_c} \times 100\%$$

where, $c_t$ means the number of covered transitions; $t_c$ means the number of all transitions. With the simulation results, the test coverage rate in each fault situation is showed 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:

$$\frac{t_c}{TC}$$

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

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

$$DCTM_a = \frac{t_c}{TC_{all}} = 350/1356 = 0.258 \text{.}$$

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

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](image-url)
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 \( t_5 \) in \( t_0 \) fault situation. Non preamble sequence can lead the state8 of FSM, so \( t_5 \) 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.

### 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


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.