Front. Electr. Electron. Eng. China 2007, 2(3): 273–276
DOI 10.1007/s11460-007-0050-7

RESEARCH ARTICLE

ZHAO Baohua, JIANG Zhenhai

A new method for improving efficiency of protocol testing

Abstract Traditional solutions have encountered some bottleneck in improving the efficiency of protocol testing. A novel method that records the test sequence dynamically is proposed. Three dynamically reordering algorithms are brought forward in line with different fault conditions. The impact of the new method of testing efficiency is also presented. Simulation results demonstrate that the proposed solution is better than the traditional ones in terms of testing efficiency.

Keywords conformance testing, test sequence dynamic reordering, testing efficiency

1 Introduction

Improving testing efficiency is difficult in protocol conformance testing. Existing testing methods are based on automata model like the finite state machine (FSM), Petri Net, etc. However, the main factor that affects testing efficiency is the test sequence generation algorithm. Currently, there are some mature generation algorithms like the T-method, UIO-method, D-method, W-method and so on. The UIO method is widely researched and applied because it generates shorter test sequences and displays stronger fault diagnosis ability. Increasing testing efficiency simply from the perspective of a test sequence generation algorithm has met some obstacles.

On the other hand, the execution order also has an effect on testing efficiency to a certain extent. However, this problem is rarely mentioned. In the existing testing methods, test sequences are tested in a static order arranged in advance. However, under some circumstances, the test does not run each test sequence necessarily. When the test aims to find the first failed test case, the cost contains only the cost of executing the test sequences before the first test case fails.

Translated from *Journal of University of Science and Technology of China*, 2006, 36(8): 882–886 [译自: 中国科学技术大学学报]

ZHAO Baohua (✉), JIANG Zhenhai
Department of Computer Science and Technology, University of Science and Technology of China, Hefei 230027, China
E-mail: bhzhao@ustc.edu.cn

Obviously, for each specified fault, there exists an optimal order, so that the first fault comes out at its earliest moment. This fact will lead to different schedulings for different faults, but we have no idea which and what fault will come out first. From the viewpoint of the average effect, any kind of static ordering method will not lead to the minimal testing cost. Thus, we propose the dynamic ordering method, that is, not to set the testing order in advance but to adjust and arrange the testing order of the left test subsequences according to the testing result of the test subsequences that have already been tested.

In this paper, we are only concerned with the output fault and transfer fault [1] and propose three dynamic ordering algorithms. The remainder of the paper is organized as follows. In Sect. 2, we introduce the algorithm when the state machine contains multiple faults. Section 3 presents the algorithm aimed at a single fault. Section 4 introduces the algorithm by joining the greedy strategy and the overall optimal strategy. The experimental results and the analysis of the computational complexity are given in Sects. 5 and 6. Finally, Sect. 7 contains the conclusions.

2 Dynamic ordering strategy for multiple faults

Definition 1 Transfer length ratio: the amount of different transfers that the test subsequence passed / length of test subsequence.

Obviously, ratio $\in (0, 1]$. To enable the first test subsequence to detect as many faults as possible and provide more information, we arrange the test subsequence that has the maximum ratio value or the longest length when they have the same ratio as the first test sequence.

Definition 2 Fault in test subsequence: the actual test result of the test subsequence is not consistent with the expectation.

In most cases, when faulty transitions are included in the test subsequence, the subsequence will fail. Exception happens when the fault is a transfer fault and the faulty transitions are the last transfer of the subsequence. When the test subsequence tc_k passes the test, we simply assume that all the transfers are correct apart from the last transfer. If the last

transfer appears many times in the subsequence, then we assume that it is correct, too. We put all the transfers that are supposed to be correct into the set called CrtTrans. When arranging the next test subsequence, we find the one that is the most diverse from CrtTrans. It needs to meet the following conditions.

1) The subsequence has the biggest amount of transfers that are not in CrtTrans. The number is denoted as differ_trans.

2) When two subsequences have the same differ_trans, we choose the one that has the shorter length.

The algorithm is given as follows.

Algorithm 1 Initial value: $\mathrm{TS}=\{tc_1,tc_2,...,tc_m,\mathrm{CrtTrans}=\{\varnothing\}$.

Step 1 Find tc_k, which is to be the first test subsequence tc_1' in TS. tc_k meets the following requirement: tc_k .ratio $\geqslant tc_j$ .ratio and if tc_k .ratio $= tc_j$ .ratio, then tc_k .len $\geqslant tc_j$ .len, $\forall tc_j \in \mathrm{TS}$.

Step 2 If tc_i' passes the test, then $\mathrm{TS}=\mathrm{TS}-\{tc_i'\}$.

Step 2.1 If $t'_{i,mi}$ appears only once in tc_i', then $\mathrm{CrtTrans}=\mathrm{CrtTrans}+\{t'_{i,1},t'_{i,2},...,t'_{i,mi-1}\}$; otherwise, if $t'_{i,mi}$ appears more than once in tc_i', then $\mathrm{CrtTrans}=\mathrm{CrtTrans}+\{t'_{i,1},t'_{i,2},...,t'_{i,mi-1},t'_{i,mi}\}$.

Step 2.2 Find tc_k, which is to be the next test subsequence tc'_{i+1} in TS. tc_k meets the following requirement: tc_k .differ_trans $\geqslant tc_j$ .dffer_trans and if tc_k .differ_trans $= tc_j$ .dffer_trans, then tc_k .len $\leqslant tc_j$ .len, $\forall tc_j \in \mathrm{TS}$.

Step 3 Stop if tc_i' fails the test.

In the algorithm, tc_k .ratio stands for the transfer length ratio of tc_k, tc_k .len stands for the length of tc_k, and tc_k .differ_trans stands for the differ_trans of tc_k.

The algorithm uses only the transfer information contained in the test subsequence and is applicable to the situation when the state machine contains multiple faults.

3 Dynamic ordering strategy for single fault

In this section, we assume that the state machine contains only one single fault.

Definition 3 Fault injection: to make a correct state machine contain one or more faulty transitions.

According to the number of fault transfers, Fault Injection is divided into two classes: Single Fault Injection and Multiple Fault Injection. After being injected with a single fault, if one test subsequence fails the test, we say this test subsequence is able to detect the fault.

Definition 4 The set of faults that could be detected by the test subsequence indicates that when the state machine is injected with all the single faults, the set of all the faults that could be detected by the test subsequence tc_k is denoted as tc_k .faults.

Generate the set of faults of all test sequences. Assuming that the state machine M has n states, m test subsequences, o outputs, then there are $f=m(o-1)+m(n-1)$ kinds of faults in total. Number each fault according to $1\rightarrow f$. For each fault, run the test subsequence tc_i, if tc_i fails the test, then add the fault number into tc_i .faults.

Arrange the test subsequence that has the biggest set of faults and shorter length to be tested first.

Denote all the test subsequences that pass the test as $\{tc_1', tc_2', ..., tc_i'\}$, and let $\mathrm{NoFaultSet}=\bigcup_{j=1}^{i} tc_j'.\mathrm{faults}$ denote no fault to exist in this set. Find tc_k in all set of test subsequences untested so that tc_k.faults and NoFaultSet are the most different. Also, tc_k needs to satisfy the following: set $tc_k.\mathrm{differFS}=|tc_k.\mathrm{faults}\text{-}\mathrm{NoFaultSet}|$, $tc_k.\mathrm{differFS}\geqslant tc_k.$ diferFS, if $tc_k.\mathrm{differFS}=tc_j.\mathrm{differFS}$, then tc_k .len $\geqslant tc_j$ .len.

The algorithm is given as follows.

Algorithm 2 Initial value: $\mathrm{TS}=\{tc_1, tc_2,...,tc_m\}$, NoFaultSet $=\{\varnothing\}$.

Step 1 Generate tc_{k_j}.faults for each test subsequence tc_k.

Step 2 Find tc_k, which is to be the first test subsequence tc_1' in TS. tc_k meets the following requirements: $|tc_k.\mathrm{faults}|\geqslant|tc_j.\mathrm{faults}|$, and if $|tc_k.\mathrm{faults}|=|tc_j.\mathrm{faults}|$, then $tc_k.\mathrm{len}\leqslant tc_j.\mathrm{len}, \forall tc_j\in \mathrm{TS}$.

Step 3 If tc_i' passes the test, then $\mathrm{TS}=\mathrm{TS}-\{tc_i'\}$, $\mathrm{NoFaultSet}=\bigcup_{j=1}^{i} tc_j'.\mathrm{faults}$. Find tc_k, which is to be the next test subsequence tc'_{i+1} in TS. tc_k meets the following requirements: $tc_k.\mathrm{differFS}\geqslant tc_j.\mathrm{differFS}$, and if $tc_k.\mathrm{differFS}=tc_j.$ differFS, then $tc_k.\mathrm{len}\leqslant tc_j.\mathrm{len}, \forall tc_j\in \mathrm{TS}$.

Step 4 Stop if tc_i' fails the test.

4 Dynamic reordering strategy by joining greedy strategy and overall optimal strategy

For the greedy strategy, study the test subsequence to be tested first so that the fault is most likely to be detected. A defect of the greedy strategy lies in that it may not obtain the optimal solution. Relatively, the overall optimal strategy evaluates all the static strategies and gets the optimal one for testing. In the following, we demonstrate the algorithm to find the overall optimal strategy.

Generate all test subsequences c_{ij} and r_{ij} for each of all the faults. Inject different faults according to the order $1\rightarrow f=m(o-1)+m(n-1)$; simulate the state machine with each fault in the order $\{tc_1,tc_2,...,tc_m\}$; record whether each test subsequence passes the test under different state machines with each fault, and record the transition number before the test fails. Denote them with r_{ij} and c_{ij}. r_{ij} stands for whether tc_j fails with fault i, that is, 1 means pass, 0 means fail. $c_{ij}=\begin{cases} tc_j.\mathrm{len}, & \text{if } r_{ij}=1 \\ k, & \text{if } r_{ij}=0,\ t_{j,k} \text{ is the first fault in } tc_j \end{cases}$, $i\in[1,f], j\in[1,m]$. List all r_{ij} and c_{ij} in Table 1 as follows.

Make all permutations with r_{ij} and c_{ij} according to row $1\rightarrow m$, then there are $m!$ static strategies in all. For permutation k, the sum of all c_{ij} before the first $r_{ij}=0$ in line i indicates the transition number before the first fault in permutation k

Table 1 c_{ij} and r_{ij}, $i \in [1,f], j \in [1, m]$

c_{ij}, r_{ij}	$j=1$	$j=2$	…	$j=m$
$i=1$	c_{11}, r_{11}	c_{12}, r_{12}	…	c_{1m}, r_{1m}
$i=2$	c_{21}, r_{21}	c_{22}, r_{22}	…	c_{2m}, r_{2m}
⋮	⋮	⋮	⋮	⋮
$i=f$	c_{f1}, r_{f1}	c_{f2}, r_{f2}	…	c_{fm}, r_{fm}

with fault i, which is denoted as avg_{ik}. $\text{avg}_{ik} := \sum_{j=1}^{l} c_{ij}, r_{i1} = r_{i2} = \ldots = r_{il-1} = 1,\ r_{il} = 0,\ \forall\ k \in [1,\ m],\ i \in [1,\ f]$

Denote avg_k as the average test cost of the static strategy k, when the test with the condition of all single faults encounter the fault, $\text{avg}_k := \sum_{i=1}^{f} \text{avg}_{ik} / f,\ k \in [1,\ m!]$. Let $\text{avg} = \sum_{k=1}^{m!} \text{avg}_k / m!$, which means the average test cost of test sequences $\{tc_1, tc_2, \ldots, tc_m\}$ when the test encounters the fault, under the situation of all static scheduling and all single faults. Let $\text{avg}_{\min} = \min\{\text{avg}_k, k \in [1,\ m!]\}$, which expresses the minimal test cost of the static scheduling. The corresponding static strategy is called optimal static scheduling strategy.

Obviously, the time complexity of finding the optimal test strategy is $m!$. The research has pointed out that for a computer that can do millions of operations per second, even when $m = 20$, the time for finding the optimal solution will last more than 700 centuries. So it is impractical to find an optimal solution.

In order to utilize the advantage that the optimal static strategy has of having a high testing efficiency, we combine the greedy strategy and the overall optimal strategy so that we can obtain higher testing efficiency.

First, find the redundant test subsequence. If the fault set of any test subsequence can be contained in some other test subsequences under the condition of single fault, then the test subsequence does not need scheduling. That is, in the set $\{tc_1.\text{faults}, tc_2.\text{faults}, \ldots, tc_m.\text{faults}\}$, find k test subsequences $\{tc'_1, tc'_2, \ldots, tc'_k\}$, so that $\bigcup_{i=1}^{k} tc'_i.\text{faults} = \bigcup_{i=1}^{m} tc_i.\text{faults}$, and k should be the least. This problem boils down to the classical minimal set cover problem, a problem that has been proved to be a nondeterministic polynomial-time hard (NP-Hard) problem. Use the set-covering heuristic function (SCHF) algorithm [6] to find the redundant test subsequence approximately. The algorithm is described as follows.

Initial value, $\text{Cover} = \{\varnothing\}, \text{Cover0} = \{S_1, S_2, \ldots, S_m\}, S_i = tc_i.\text{faults}$

Step 1 Find S_{i0} s.t. $F(S_i)$ is maximal, find S_{j0} s.t. $F(S_i)$ is minimal ($S_i, S_{i0}, S_{j0} \in \text{Cover0}$).

Step 2 If $F(S_{i0}) > L$, then $\text{Cover} = \text{Cover} + \{S_{i0}\}$, $S = S - S_{i0}$, $\text{Cover0} = \text{Cover0} - \{S_{i0}\}$, else $\text{Cover0} = \text{Cover0} - \{S_{i0}\}$.

Step 3 If $S = \varnothing$, then stop after outputting Cover, or else jump to Step 1.

In the algorithm, $F(S_i) = \dfrac{1 + R(S_i)}{P(S_i) - 1} + \dfrac{1}{N - |S_i|}$, $N = |S|$, $L = N + 2$, $P(S_i)$ is the degree of the coverage of S_i, $R(S_i)$ is the degree of S_i [6]. The set Cover output by the algorithm is the set that needs dynamic scheduling. The other test subsequences are put in the end and not contained in the scheduling.

Then choose k test subsequences, put them in the front and arrange them with static scheduling. Find k test subsequences in the set Cover and the union set of the fault set of these k test subsequences is maximal. Test these k test subsequences first. In these k test subsequence, put the shorter with the bigger fault set in front. When doing the test, these k test subsequences are tested in static order. Here, how to choose k will have an impact on the testing efficiency. According to experiments, generally, $k \in [m/4,\ m/3]$ and $k \leqslant 5$ is preferred. We can also use algorithm SCHF to find these k test subsequences, while the only difference is that the stop condition in Step 3 is changed to $|S| = m - k$. The algorithm is described as follows.

Algorithm 3 Initial value: $\text{TS} = \{tc_1, tc_2, \ldots, tc_m\}$, $\text{Cover} = \{\varnothing\}$.

Step 1 Generate $tc_{k_j}.\text{faults}$ for each test subsequence tc_k.

Step 2 Generate the set of test subsequence to be scheduled, according to SCHF algorithm, denote the set as Cover.

Step 3 According to the modified SCHF algorithm (The stop condition in Step 3 is changed to $|S| = m - k$), find k test subsequences $\{tc'_1, tc'_2, \ldots, tc'_k\}$, subject to $\left|\bigcup_{i=1}^{k} tc'_i.\text{faults}\right| \geqslant \left|\bigcup_{i=1}^{k} tc_{ji}.\text{faults}\right|$, $\forall tc_{ji} \in \text{TS}$, in which $|tc'_i.\text{faults}| \geqslant |tc'_{i+1}.\text{faults}|$ and if $|tc'_i.\text{faults}| = |tc'_{i+1}.\text{faults}|$, then $tc'_i.\text{len} \leqslant tc'_{i+1}.\text{len}$. Test these k test subsequences in static order. Here, $k \in [m/4, m/3]$ and $k \leqslant 5$.

Step 4 From the $(k+1)$ th test subsequence tc'_{k+1}, schedule according to Algorithm 2.

5 Experimental results

The following shows the experimental results of using the three dynamic scheduling algorithms onto the state machine (Fig. 1):

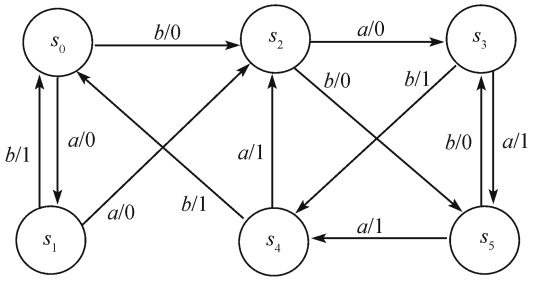

Fig. 1 FSM for experiment

The regular static scheduling is the scheduling strategy in practical application.

To further validate the effectiveness of the proposed dynamic strategy in practice and also in a big state machine,

Table 2 Experimental results of Fig. 1

Item	Regular static scheduling	Static optimal scheduling avg_{min}	Average static scheduling avg	Dynamic scheduling (Algorithm 1)	Dynamic scheduling (Algorithm 2)	Dynamic scheduling (Algorithm 3)
Single fault	16.52	11.43	16.32	14.45	12.94	11.74
Two faults	9.95	8.04	9.85	7.76	…	…
Three faults	7.01	6.50	6.89	5.38	…	…

Table 3 Experimental results on BGP FSM

Item	Regular static scheduling	Static optimal scheduling avg_{min}	Average static scheduling avg	Dynamic scheduling (Algorithm 1)	Dynamic scheduling (Algorithm 2)	Dynamic scheduling (Algorithm 3)
Single fault	28.21	Null	Null	20.65	17.54	14.37
Two faults	18.35	Null	Null	13.23	…	…
Three faults	13.58	Null	Null	9.76	…	…

we test the border gateway protocol (BGP) FSM by employing the dynamic strategy. The BGP is an external gateway protocol for communication among ASes and is applied widely in practice. The results of the experiment are shown in Table 3.

Because BGP possesses 22 testing subsequences, we cannot find the value of optimal scheduling method (avg_{min}) and the average static scheduling method(avg). We can also see that the dynamic scheduling improves the test efficiency greatly compared to the regular static scheduling, and is better than the average value of the static scheduling and approximate to that of the static optimal scheduling.

6 Complexity analysis

Now let us analyze the computational complexity of the most complex Algorithms 1–3. Consider that the state machine M has n states, m edges, m' testing subsequences, ($m' \leqslant m$) and o outputs. The first step, generating the fault set, has time complexity $O\left((o+n-2)mm'\right)=O\left((o+n-2)m^2\right)$; the second step, finding redundant testing subsequences, has time complexity $O\left(mm'^2+m'^3\right)=O\left(m^3\right)$; the third step, seeking k static scheduler subsequences, has time complexity $O(m^3)$. After each testing subsequence has been carried out, the subsequences following the dynamic scheduling has a time complexity of $O\left(m'\right)=O\left(m\right)$, therefore the total time complexity of Algorithm 3 is $O\left((o+n-2)m^2\right)+O\left(m^3\right)+O\left(m^3\right)+O\left(m\right)=O\left(m^3\right)$.

It can be concluded that, the time complexity of Algorithm 3 is mostly spent in seeking redundant testing subsequences and generating fault set. These tasks are the preparation before the actual testing and exert no influence on the dynamic scheduling of the actual testing. Thus, the time for scheduling in testing is comparatively smaller so that we can derive higher testing efficiency with less cost.

7 Conclusions

This paper is based on the mature research of the algorithm of generating a test sequence. From the perspective of scheduling strategy of a testing sequence, it proposes a novel method that dynamically reorders the test sequences. The paper also puts forward three different dynamical reordering algorithms. According to the practical testing, the method reduces the testing cost and increases the efficiency while taking a shorter time. We can conclude that the method is practical.

Acknowledgements This work was supported by the National Natural Science Foundation of China (Grant No. 60241004 and 60602016), the National Basic Research and Development Program of China (No. 2003CB314801), MOE-MS Key Laboratory of Multimedia Calculation and Communication Open Foundation (No. 05071801), and Huawei Foundation (No. YJCB2006044TS).

References

1. Myungchul Kim, Sangjo Yoo, Jinhee Park, et al. A dynamic protocol conformance test method. Journal of Systems and Software, 2003, 67(1): 31–43
2. Pomeranz I, Reddy S M. Sequence reordering to improve the levels of compaction achievable by static compaction procedures. In: Proceedings of Conference and Exhibition on Design Automation and Test in Europe. Piscataway, NJ: IEEE Press, 2001, 214−218
3. Naik K. Efficient computation of unique input/output sequences in finite state machines. IEEE/ACM Transactions on Networking, 1997, 5(4): 585–599
4. Lai R. A survey of communication protocol testing. Journal of Systems and Software, 2002, 62(1): 21–46
5. Bochmann G V, Das G, Dssouli A, et al. Fault models in testing. In: Proceedings of IWPTS IV-International Workshop on Protocol Tests Systems. Leischendam, The Netherlands, 1991, 17–30
6. Quan Riguang, Hong Bingrong, Ye Feng, et al. A heuristic function algorithm for minimum set-covering problem. Journal of Software, 1998, 9(2): 156–160