DISTRIBUTED TEST USING LOGICAL CLOCK

Young Joon Choi[1], Hee Yong Youn[2], Soonuk Seol[1], and Sang Jo Yoo[3]
[1]*School of Engineering, Information and Communications University*
{cyjnice, suseol}@icu.ac.kr
[2]*School of ECE, Sungkyunkwan University, Suwon, Korea*
youn@ece.skku.ac.kr
[3]*The Graduate School of Information Technology & Telecommunications, Inha University, Inchon, Korea*
sjyoo@inha.ac.kr

Abstract It is difficult to test a distributed system because of the task of controlling concurrent events. Existing works do not propose the test sequence generation algorithm in a formal way and the amount of message is large due to synchronization. In this paper, we propose a formal test sequence generation algorithm using logical clock to control concurrent events. It can solve the control-observation problem and makes the test results reproducible. It also provides a generic solution such that the algorithm can be used for any possible communication paradigm. In distributed test, the number of channels among the testers increases non-linearly with the number of distributed objects. We propose a new remote test architecture for solving this problem. SDL Tool is used to verify the correctness of the proposed algorithm, and it is applied to the message exchange for the establishment of Q.2971 point-to-multipoint call/connection as a case study.

Keywords: concurrent events, control-observation problem distributed testing, logical clock, output-shifting faults, test sequence

1. INTRODUCTION

Distributed objects have been implemented by Open Distributed Processing (ODP) [11] and others. ODP is an ISO/ITU standard, offering a generic framework for the development of distributed system. In this paper, distributed system developed based on ODP is taken as the target [1] to test.

Benattou et al. [1] proposed a method to test distributed system as a black box using a distributed test method. Here extra signals were proposed to be used among the testers through coordination channels, called multicast channel, for synchronizing them. The method has a merit of making the events occurred at the distributed objects be totally ordered with special signals, but has a drawback of possible output-shifting faults [8]. A method in [2] is to use synchronization messages for controlling concurrent events as a grey box. However, it has a difficulty in choosing a way to observe the internal status of implementation under test (IUT) and needs other messages in addition to TS. Luo et al. [3] analyzed existing models based on synchronized TS and surveyed fault coverage in terms of output, transfer, and hybrid. However, it did not propose any Test Sequence (TS) generation algorithm in a formal way.

In overall, existing works for controlling concurrent events in the test of distributed system do not propose the algorithm in a formal way and increase the amount of message within IUT due to synchronization. To overcome the drawbacks, we propose a TS generation algorithm using logical clock that formally generates signals to control concurrent events by numerically labeling the events of TS with logical clock values and comparing them. It can also solve the control-observation problem [1,3]. The proposed TS generation algorithm based on logical clock allow the test results to be reproduced since a totally ordered TS can be achieved with the controlling of concurrent events.

In distributed test, several testers are used and synchronized by exchanging coordination messages. While traditional distributed test method is effective in testing IUT of multiple ports, it has a shortcoming that the number of channels among the testers increases non-linearly with the number of distributed objects. To overcome this drawback, we also propose an approach of remote test architecture. Specification and Description Language (SDL) Tool [14] is used to verify the correctness of the proposed algorithm, and it is applied to the message exchange for the establishment of Q.2971 point-to-multipoint call/connection as a case study.

The rest of the paper is structured as follows. Section 2 explains the background on testing distributed system. The proposed TS generation algorithm based on logical clock is proposed in Section 3. Section 4 applies the algorithm to the remote test architecture, and finally Section 5 concludes the paper with some future research work.

2. BACKGROUND

When a distributed system is tested, some of the events executed by the distributed objects do not always have casual relationship but possibly

concurrent relationship with each other. If the events are concurrent, the following **control-observation problem** may take place [1, 3].

- When concurrent inputs are applied to a black box of a form of IUT, the test results depend on the order of input process in the IUT.
- In testing a black box IUT, even though the mapping of an input and its output is incorrect, test results may be correct.

The events occurred at each distributed object, which is an element of a distributed system, may have concurrent relationship with other events. In this case, the test results can be different from each other when an IUT is tested with the same TS repeatedly since all the events are not totally ordered. Therefore we need a mechanism to make concurrent events have causal relation in order to reproduce the same test results. Causal relation is also called "happened before" relation, represented by '$\rightarrow$'. Relation 'a' $\rightarrow$ 'b' should satisfy one of the following rules to be 'happened before' events [5, 6].

Rule 1
(1) 'a' and 'b' are in the same distributed object and 'a' $\rightarrow$ 'b'.
(2) 'a' is sender and 'b' is receiver. However, in case of synchronous communication, both send and receive events occur at the same time.
(3) 'a' $\rightarrow$ 'c' and 'c' $\rightarrow$ 'b'.

Logical clocking is a time-stamping mechanism, which can be represented with linear time, vector time, and matrix time [5]. In this paper, we adopt vector time describing logical clock in one-dimensional matrix whose field is assigned to each object [5]. The value of a field of logical clock denotes the time the corresponding event occurs at a distributed object. With the vector time, the following rule should be satisfied so that two events can be causally related.

Rule 2
If the following relation is satisfied between event 'X' and 'Y' in an IUT consisting of distributed objects a, b and c, the two events are said to have a causal relation "a $\rightarrow$ b" except a case as ($X_a = Y_a$, $X_b = Y_b$, $X_c = Y_c$). The logical clock values of events 'X' and 'Y' are (X_a, X_b, X_c) and (Y_a, Y_b, Y_c), respectively.

$$(X_a \leq Y_a, X_b \leq Y_b, X_c \leq Y_c)$$

A stable state means a global state, which is not changed into another state without any input [7]. In testing a black box IUT, it is difficult to observe the transient states and only the stable states are visible.

Figure 1. Stable states and transient states.

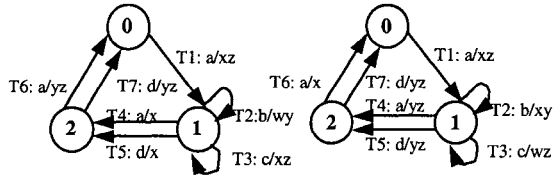
Figure 1 represents a finite state machine (FSM), which reaches stable state-3 via transient state-1 and 2 after input 'a' is applied at stable state-0. A tester can infer the test result by observing state-0 before the input 'a' was applied and state-3 after the output 'x' and 'z' are obtained. This allow to reduce the number of states which should be managed in the test and also avoid the difficulty of observing transient states.

3. THE PROPOSED ALGORITHM

In this section we first analyze the existing method proposed for controlling concurrent events in a black box IUT. We then propose a TS generation algorithm based on logical clock.

3.1 The Problems in Existing Test Method

As a case study, we consider a typical test method reported in [1]. Figure 2(a) represents a FSM M, where each state is stable. The FSM M is a 3-port FSM whose inputs and outputs at each port are shown in Table 1. Figure 2(b) is faulty FSM M derived from Figure 2(a). Here each port is connected to a tester whose identifier is the same as the port.



(a) FSM M (b) Faulty FSM M

Figure 2. 3-port FSM.

Table 1. Inputs and outputs at each port of Figure 2.

	port 1	port 2	port 3
Input	a	b	c, d
Output	x, w	y	z

Definition 1: Transition T is represented by $(s_i, s_j, x/y)$, where $x \in X$, $y \in Y$, $\{s_i, s_j\} \subset S$, $s_j = \delta(s_i, x)$, and $y = \lambda(s_i, x)$. X is a finite set of input symbols, Y is a finite set of output symbols, S is a finite set of states, δ is a state transfer function, and λ is an output function.

If an IUT implemented with a faulty FSM M is tested by a TS (T1 → T3 → T2 → T5 → T6), the control-observation problem may occur in the subsequence T3 → T2 and T5 → T6. None of the testers can detect the faults if 'b' is read first by the IUT when the input 'c' of transition T3 and the input 'b' of transition T2 are applied to the IUT concurrently in state-1. It is because the order of the outputs observed by the testers is correct even though the mapping of the input and its output is incorrect in the subsequence T5 → T6.

The TS of a distributed object usually consists of inputs and outputs, and signal 'C' and 'O' controlling concurrent events. The signal 'C' gives casual relation to concurrent inputs applied to IUT, while signal 'O' gives causal relation between concurrent input and output. The following steps produce signal 'C' and 'O'. A new TS for each distributed object including signal 'C' and 'O' is called Local Test Sequence (LTS).

1. Label logical clock values for the events of TS.
2. Classify the events into either causal or concurrent by comparing the logical clock values of the events.
3. If there exist concurrent events, generate signal 'C' and 'O' which make the concurrent events be causally related and insert them to the TS of each distributed object.

To solve the control-observation problem, [1] proposed coordination channels whose function is to send or receive signal 'C' and 'O' among the testers. However, output–shifting fault which is a consequence of the control-observation problem can occur.

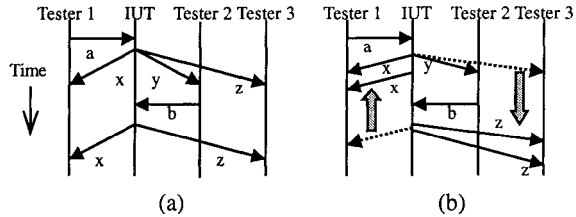


Figure 3. An example of output-shifting faults.

Refer to Figure 3(a) where an IUT is tested by TS (a/xyz → b/xz). The signal 'C' and 'O' are not generated by the method proposed in [1]. As a result, none of the testers can detect the faults even if the output 'z' of "a/xyz" is shifted forward and the output 'x' of "b/xz" is shifted backward as shown in Figure 3(b). This is because the order of the input and **output** observed by each tester is the same as Figure 3(a). This is called output-**shifting faults** [8]. Output-shifting faults are the case that the test results are correct even though the mappings of input and output are incorrect. This occurs since it cannot properly control the testers receiving the outputs of the

previous transition which has concurrency relation with the new input. A new formal way for signal generation is thus needed to be developed.

3.2 The Proposed Local Test Sequence Generation Algorithm

Here we propose an LTS generation algorithm which can solve the control-observation problem including output-shifting faults in a formal way using logical clock. We make assumptions as follows:

- IUT is a black box.
- The logical clock vector is represented as a format (tester 1, IUT, tester 2, tester 3). When an event occurs at an element, the value of the corresponding field is increased.
- The output receive delay for a transmission is zero.
- Logical clock is increased by the unit time.

Figure 4 shows TS1 (T1→ T3→ T2 → T4 →T7) which tests the IUT implemented with FSM M. Observe that each event is labeled with a logical clock value. To control concurrent events classified by Rule 2, extra signals are needed. Table 2 lists the comparisons of logical clock values labeled with the events of subsequence T1→ T3 and T3 → T2. Here Compare_LC(x, y) is a function that returns 'true' if event 'x' and 'y' satisfies Rule 2. Otherwise, it returns 'false'.

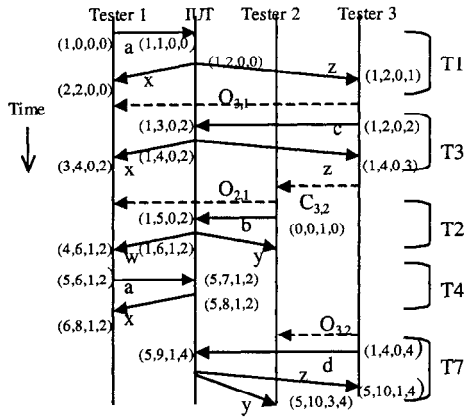


Figure 4. Logical clock labeling at the events of Test Sequence 1 (TS1).

Table 2. Comparison of logical clock values.

T1 (0, 1, a/xz) →T3 (1, 1, c/xz)	T3 (1, 1, c/xz) →T2 (1, 1, b/wy)
Compare_LC(a, c) = true	Compare_LC(c, b) = false
Compare_LC(x, c) = false	Compare_LC(x, b) = false
Compare_LC(z, c) = true	Compare_LC(z, b) = false

According to Table 2, only output ‘x’ of transition T1 and input ‘c’ of transition T3 are concurrent in subsequence T1 → T3. In this case, a signal making input ‘c’ and output ‘x’ be casually related is needed. This signal is called signal ‘O’. Accordingly, Tester 3 that provides input ‘c’ of transition T3 to IUT sends signal ‘$O_{3,1}$’ to Tester 1, which receives output ‘x’ of transition T1 . Similarly, input ‘b’ of transition T2 is concurrent with all the inputs and outputs of transition T3 in subsequence T3 → T2. Therefore a signal is needed to let input ‘b’ be applied to the IUT after transition T3 occurs. This signal is called signal ‘C’. In this case, Tester 3 that is one of the testers receiving the outputs of transition T3 sends signal ‘$C_{3,2}$’ to Tester 2, which applies input ‘b’ of transition T2 to the IUT. The signal ‘$C_{3,2}$’ suggests that Tester 3 had already received output ‘z’. However, Tester 2 sends additional signal ‘$O_{2,1}$’ to Tester 1 since input ‘b’ is still concurrent with output ‘x’ of transition T3.

None of the testers can detect faults even though output ‘y’ is backward-shifted since the test scenario of Tester 2 is the same as that for correctly implemented FSM being tested by subsequence T4 → T7. To solve this problem, Tester 3 sends signal ‘$O_{3,2}$’ to Tester 2 before Tester 3 applies input ‘d’ to the IUT.

As aforementioned, the output receive delay is assumed to be zero, and thus each tester receives the output of a transition instantly. Therefore, signal ‘O’ controlling concurrent outputs is generated by the tester which sends input to the IUT. This assumption was also adopted in [1]. With this assumption, though, the TS generation algorithm has some limitations in testing real distributed system.

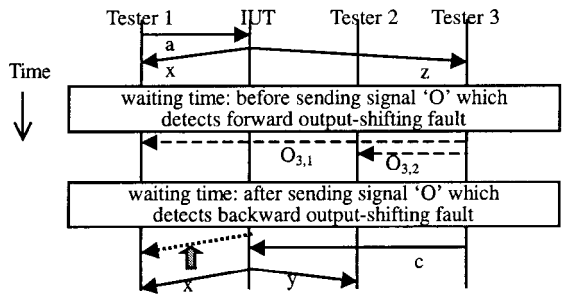


Figure 5. Waiting time and over-waiting time problem.

For example, refer to Figure 5. Tester 1 decides that the IUT is faulty when it receives output ‘x’ in “a/xz” after Tester 1 eceives signal ‘$O_{3,1}$’ even if the IUT is correctly implemented. Also Tester 2 cannot detect faults in case that output ‘y’ in “c/xy” is backward-shifted. This is because synchronization among the testers is done through synchronization signal, but IUT cannot be directly synchronized with the testers. Accordingly, the testers should have sufficient waiting time to synchronize with the IUT.

However, if Tester 1 receives backward-shifted output ‘x’ in “c/xy” after some waiting time, it is impossible to detect the fault. Let us call this **over-waiting problem.** In order to prevent the problem, the waiting time and receive delay time of outputs for a transmission are assumed to be zero. The related definitions and the proposed algorithm are presented as follows.

Definition 2

- In transition T = (si, sj, x/y), “x/y” stands for “!x?y”.
- ‘-’ means that signal is sent to the tester and ‘+’ means that signal is received from other testers.
- Port(x) is a function that returns ‘port id’ generating input or output ‘x’.
- PortS(X) = {Port(x1), Port(x2), . . . , Port(xn)}, where X = {x1, x2, . . . , xn} and xi is an event.
- Compare_LCS(x, Y) is a function that returns false if Compare_LC(x, a) = false for all a ∈ Y, and return true otherwise where Y is a set of input or output.
- Label_LC(x, k) is a function that updates logical clock value of port k and records logical clock value of x where x is an input or output.
- Label_LCS(X, PortS(X)) = {Label_LC(x1, Port(x1)), Label_LC(x2, Port(x2)), . . . , Label_LC(xn, Port(Xn))}. Append(X, Y) is a function concatenating string X and Y.
- Append_In (Port(x), x) = Append(LTSPort(x), Append(“!”, x)) where LTSPort(x) is an LTS of the tester connected to Port(x).
- Append_Out(PortS(X), X) = {Append(LTSPort(X1), Append(“?”, “x1”)), Append(LTSPort(X2), Append(“?”, “x2”)), . . . , Append(LTSPort(Xn), Append(“?”, “xn”))}.

ALGORITHM_LTS /* LTS GENERATING ALGORITHM */
1: INPUT: TS = $!s_1?Y_1 \ldots !s_m?Y_m$ /*The elements of Y_i can be more than zero and is represented with an array */
2: OUTPUT: local test sequences = (LTS_1, … ,LTS_n) /* n is the number of distributed objects/
3: Send_Set = null /* Send_Set is a set of testers receiving signal ‘O’ */
4: **for**(i = 1; i ≤ m; i++) /* m is the number of transitions */
5: Label_LC(s_i, Port(s_i));
6: Label_LCS(Y_i, PortS(Y_i)); <------ /* Logical clock values labeling at input s_i and outputs Y_i */
7: **end for**
8: **for**(i = 1; i ≤ m; i++)
9: k = Port(s_i); h = Port(s_{i+1});
10: Append_In(Port(s_i), s_i);
11: **if** i < m
12: **case** $|Y_i| \neq 0$:
13: Append_Out(PortS(Y_i), Y_i); <-- /* Comparison of logical clock values for generating signal ‘C’ */
14: **if** (Compare_LCS(s_{i+1}, Y_i) = false) and (Compare_LC(s_{i+1}, s_i) = false)
15: select a sender (tester) which generates signal ‘C’ where sender ∈ PortS(Y_i);
16: Append(LTS_{sender}, Append(“-”, “$C_{sender,k}$”)); Append(LTS_k, Append(“+”, “$C_{sender,k}$”));
17: **end if**
18: **for**(i = 0; i < $|Y_i|$; i++) <-- /* Comparison of logical clock values for generating
19: **if** (Compare_LC(s_{i+1}, Y_i[j]) = false) and (Port(Y_i[j]) ≠ sender) signal ‘O’ to protect forward output-shifting faults */
20: Send_Set ← Send_Set ∪ Port(Y_i[j]);
21: **end if**
22: **end for** /* Comparison of logical clock values for generating
23: **for**(i = 0; i < $|Y_{i+1}|$; i++) signal ‘O’ to protect backward output-shifting faults */
24: **if** f(Compare_LCS(Y_{i+1}[j], Y_i) = true) and (Port(Y_{i+1}[j]) ∉ PortS(Y_i)) and (Port(Y_{i+1}[j]) ≠ h)] or [(Compare_LCS(Y_{i+1}[j], Y_i) = false) and (Port(Y_{i+1}[j]) ≠ h)]
25: Send_Set ← Send_Set ∪ Port(Y_{i+1}[j]);
26: **end if**

```
27:   end for
28:   case |Yi| = 0:                    /* Comparison of logical clock value for generating signal 'C' */
29:    if Compare_LC( si, si+1) = false
30:     Append(LTSk, Append("-", "Ck,h "); Append(LTSh, Append("+", "Ck,h "));
31:    end if
32:    for(j = 0; j < |Yi+1|; j++)
33:     if Port(Yi+1[j]) ≠ h            /* Port comparison for generating signal 'O' to
34:      Send_Set ← Send_Set ∪ Port(Yi+1[j]);   protect backward output-shifting faults */
35:     end if
36:    end for
37:   end case
38:   for ∀p ∈ Send_Set
39:    Append(LTSh, Append("-", "Oh,p"); Append(LTSp, Append("+", Oh,p));
40:   end for
41:  else if (i = m) and (|Yi | ≠ 0)
42:   Append_Out(Port(Yi), Yi);
43:  end if
44: end for
end Algorithm_LTS
```

In the algorithm above, Lines from 4 to 7 are to label logical clock values for the events of TS. Whenever input or output is generated from the IUT, Label_LC (or Label_LCS) function updates the logical clock value of the distributed object where the input or output is generated. Lines from 14 to 40 are to generate signal 'C' and 'O' in the transitions except the final transition. When the output of a transition is not null, the logical clock values of the events between the current transition and the input events of the next transition are compared using Line 19. If they are concurrent, signal 'C' is generated. Lines 19 and 24 are the statements comparing logical clock values for controlling forward output-shifting faults and backward output-shifting faults, respectively. When the output of a transition is null, the logical clock values between the current and next input are compared using Line 29. If they are concurrent, signal 'C' is generated. Line 33 is to solve backward output-shifting faults in case of null output by comparing the clock values. Figure 6 is a test scenario obtained as a result of applying Algorithm_LTS to TS1.

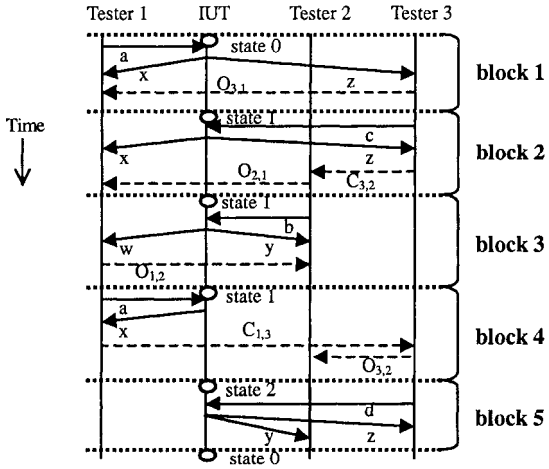


Figure 6. Applying Algorithm_LTS to TS1.

Note that state 0, 1 and 2 in Figure 6 are stable states since they are not changed to other states before a new input is applied to the IUT. If the area between the stable states is represented as a block, happened-before relationships also exist between the blocks since the events in each block are totally ordered with the events in other blocks by signal 'C' and 'O'.

3.3 Extension of the Proposed Local Test Sequence Generation Algorithm

The LTS generation algorithm [1] and the proposed Algorithm_LTS assume that the output receive delay is zero. If the assumption is not satisfied, test results may not be able to be reproduced. For example, in Figure 6, even though IUT is correct, the test results can be faulty when Tester 3 receives output 'z' and then sends signal '$O_{3,1}$' to Tester 1 before output 'x' is received by Tester 1 in the first block. To solve the problem, LTS for Tester 3 should be updated so that input 'c' can be sent to the IUT after Tester 1 sends signal '$O_{3,1}$' to Tester 3 and Tester 3 receives both signal '$O_{3,1}$' and output 'z'.

Let us call the LTS generation algorithm for which output receive delay is not zero Algorithm_LTS_DELAY. Append_RD(X, Y) is a function attaching Y to X while keeping the order of the elements of them. For example, Append_RD("xy", "z") means that 'xyz', 'xzy', or 'zxy'.

ALGORITHM_LTS_DELAY

1: INPUT: TS = $!s_1?Y_1 \ldots !s_m?Y_m$ /*The elements for Y_i can be more than zero and is represented by an array */
2: OUTPUT: local test sequences = (LTS_1, … ,LTS_n) /*n is the number of distributed objects */
3: Send_Set = null /* Send_Set is a set of testers which should receive signal 'O' */
4: **for**(i = 1; i ≤ m; i++) /* m is the number of transitions */
5: Label_LC(s_i, Port(s_i));
6: Label_LCS(Y_i, PortS(Y_i)); <-- /* Logical clock values labeling at input s_i and outputs Y_i */
7: **end for**
8: **for**(i = 1; i ≤ m; i++)
9: k = Port(s_i); h = Port(s_{i+1});
10: Append_In(k, s_i);
11: **if** i < m
12: **case** $|Y_i| \neq 0$:
13: **if** (Compare_LCS(s_{i+1}, Y_i) = false) and (Compare_LC(s_{i+1}, s_i) = false)
14: Append_Out(PortS(Y_i), Y_i);
15: select a sender (tester) which generates signal 'C' where sender ∈ PortS(Y_i);
16: Append(LTS_{sender}, Append("-", "$C_{sender,h}$"));
17: Append_RD(temp, Append("+", "$C_{sender,h}$"));
18: Temp_Set ← PortS(Y_i) \ sender; /* / is a difference set */
19: **else if** (Compare_LCS(s_{i+1}, Y_i) = false) and (Compare_LC(s_{i+1}, s_i) = true)
20: Append_Out(PortS(Y_i), Y_i);
21: Temp_Set ← PortS(Y_i);
22: **else if** Compare_LCS(s_{i+1}, Y_i) = true
23: **for**(j = 0; j < $|Y_i|$; j++)
24: **if** Compare_LC(s_{i+1}, Y_i[j]) = false
25: Temp_Set ← Temp_Set ∪ Port(Y_i[j]);
26: Append(LTS_a, Append("?", Y_i[j]));
27: **else** Append(temp, Append("?", Y_i[j]));
28: **end if**
29: **end for**

/* Case 1: comparison of logical clock values for generating signals 'C' and 'O' */

/* Case 2: comparison of logical clock values for generating signal 'O' */

/* Case 3: comparison of logical clock values for generating signal 'O' */

```
30:  end if
31:  for ∀p ∈ Temp_Set
32:   Append(LTS_p Append("-", "O_p,h")); Append_RD(temp, Append("+", "O_p,h"));
33:  end for                                   /* Comparison of logical clock values for generating signal
34:  Append(LTS_h, temp);                      'O' to protect backward output-shifting faults */
35:  for(i = 0; i < |Y_i+1|; j++)
36:   if [(Compare_LCS(Y_i+1[j], Y_i) = true) and (Port(Y_i+1[j]) ∉ PortS(Y_i)) and
      (Port(Y_i+1[j]) ≠ h)] or [(Compare_LCS(Y_i+1[j], Y_i) = false) and (Port(Y_i+1[j]) ≠ h)]
37:    Send_Set ← Send_Set ∪ Port(Y_i+1[j]);
38:   end if
39:  end for
40: case |Y_i| = 0:
41:  if Compare_LC( s_i, s_i+1) = false   <----- /* Comparison of logical clock value for generating signal 'C' */
42:   Append(LTS_k, Append("-", "C_k,h "); Append(LTS_h, Append("+", "C_k,h "));
43:  end if
44:  for(i = 0; i < |Y_i+1|; j++)          /* Port comparison for generating signal 'O' to protect backward
45:   if Port(Y_i+1[j]) ≠ h <-------------      output-shifting faults*/
46:    Send_Set ← Send_Set ∪ Port(Y_i+1[j]);
47:   end if
48:  end for
49: end case
50: for ∀p ∈ Send_Set
51:  Append(LTS_k, Append("-", "O_k,p"); Append(LTS_p, Append("+", O_k,p));
52: end for
53: else if (i = m) and (|Y_i| ≠ 0)
54:  Append_Out(Port(Y_i), Y_i);
55: end if
56: end for
end Algorithm_LTS_DELAY
```

The lines from 13 to 34 are modified from Algorithm_LTS. Line 13 (Case 1), Line 19 (Case 2), and Line 22 (Case 3) compare logical clock values for controlling forward output-shifting faults in case that the outputs are not null. Case 1 is applied when all the events of the current transition and an input of the subsequent transition are concurrent, Case 2 is applied when outputs of the current transition and an input of the subsequent transition are concurrent, and Case 3 is applied when some outputs of the current transition and an input of subsequent transition are causally related, respectively. The signal 'O' generated by each case is sent from the tester (Temp_Set), which receives the outputs of the current transition, to a tester which sends an input of the subsequent transition to the IUT. Therefore, the LTS for input-side testers should manage all combinations of the receive order of the input events. Generating signal 'O' that controls backward output-shifting faults is the same as the one in Algorithm_LTS.

Figure 7 is a sample test scenario. Here Specification and Description Language (SDL) Tool [14] is used to verify the correctness of Algorithm_LTS_DELAY. An FSM representing the IUT and the LTS obtained by Algorithm_LTS_DELAY for TS1 are described in SDL processes, respectively. Note that the output receive delay is not zero.

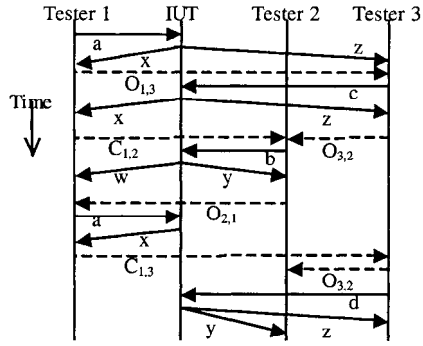


Figure 7. The LTS obtained by Algorithm _LTS_DELAY for TS1.

We analyze the results of Message Sequence Chart (MSC) Trace which is the test scenario for Figure 7 using reachability tree. Figure 8 shows the reachability tree obtained by the SDL Tool. The states represented at each point are global ones showing the states of Tester 1, IUT, Tester 2 and Tester 3, respectively.

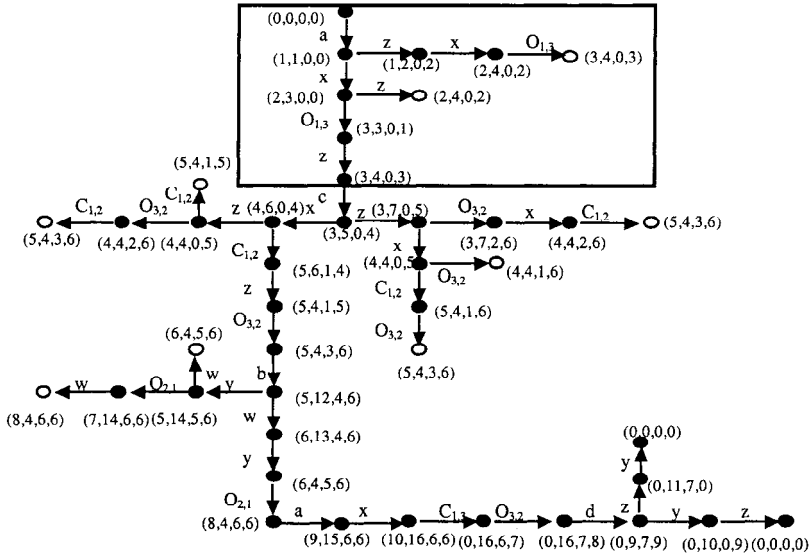


Figure 8. The reachability tree for Figure 9.

The box shown in Figure 8 represents the reachability tree for transition T1. To send input ‘c’ of transition T3 to the IUT, the logical clock value should be (3, 4, 0, 3). In other words, input ‘c’ is applied to the IUT only at stable state (3, 4, 0, 3). This demonstrates that Algorithm_LTS_DELAY can reproduce the test results by controlling concurrent events.

We apply Algorithm_LTS and Algorithm_LTS_DELAY to the message exchange for the establishment of Q.2971 point-to-multipoint call/connection. Figure 9 illustrates a message exchange where call initiator is S_A and call receivers are R_A and R_B [12, 13]. In here, the IUT has 3 ports connected to S_A, R_A and R_B.

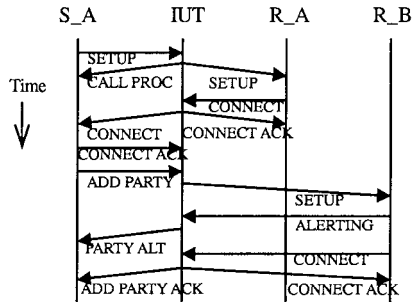


Figure 9. Message exchange for the establishment of Q.2971 network-side npoint-to-multipoint call/connection.

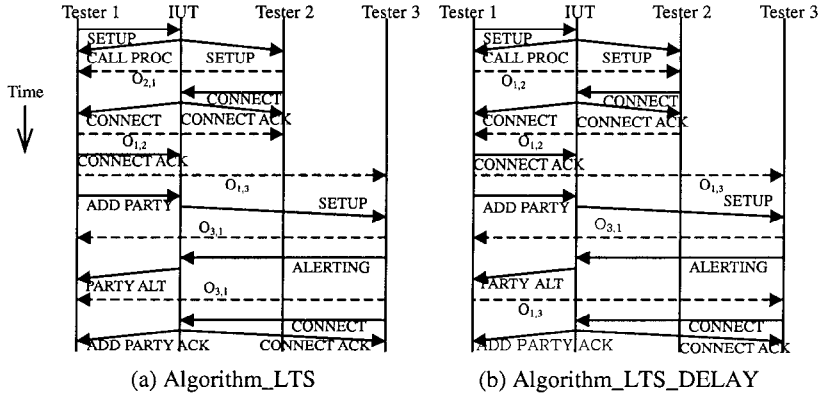


Figure 10. The LTS obtained by the two algorithms.

In Figure 10(a), the testers must receive the outputs without delay. However, message exchange in communication networks such as Q.2971 cannot satisfy such condition. The test scenario of Figure 10(b) can be said to be much more general. Table 3 compares the LTS generation algorithms: the algorithm in [1], Algorithm_LTS, and Algorithm_LTS_DELAY.

Table 3. Comparison of Local Test Sequence generation algorithms.

	[1]	Algorithm_LTS	LTS_DELAY
Output receive delay	No	No	Yes
Backward output-shifting faults	Yes	No	No
Forward output-shifting faults	Yes	No	No
Over-waiting time problem exists	-	Yes	No
Channel complexity	$O(n^2)$	$O(n^2)$	$O(n^2)$

In Table 3, Algorithm_LTS prevents output-shifting faults except over-waiting time problem. With Algorithm_LTS_DELAY, appropriate waiting time is allowed for preventing over-waiting time problem. The channel complexity of the algorithms is $O(n^2)$ since they employ a test method of mesh-structure. The numbers of signal 'C' and 'O' generated by the proposed algorithms are same, while it is slightly larger than that of [1].

4. TEST ARCHITECTURE

In this section, we propose a method for testing distributed system using Algorithm_LTS_DELAY with no coordination channels.

4.1 The Proposed Test Architecture

The number of coordination channels among the testers in distributed test environment exponentially increases since the channels are connected as a mesh. So, the diagnostic power of a tester diminishes due to the communication overhead. In order to solve the problem, a test architecture having O(n) channel complexity is proposed as Figure 11. Here Management and Control Entity (MCE) receives signal 'O' from the testers and sends the signal to them.

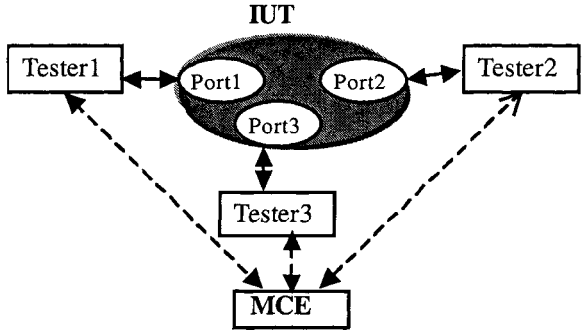


Figure 11. Proposed test architecture.

Here the transitions (state a, state b, x, y) and (state b, state c, x, y), and the tester connected to Port(x) should receive synchronization signals except cases that Port(y) = null, Port(x) = Port(x) and Port(x) = Port(y). This is because the testers are synchronized with other testers via only MCE. Therefore, only signal 'O' is used to synchronize the testers. Figure 12 shows the result obtained by applying the proposed algorithm to the Figure 9.

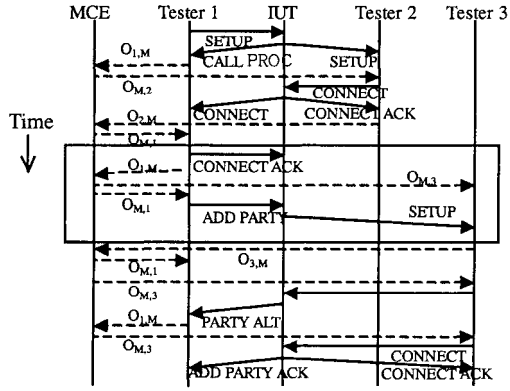


Figure 12. An LTS for message exchange of the establishment of Q.2971.

To apply the proposed algorithm, extra signals are needed between the testers and an MCE in case of the transitions having concurrent events or no output for any input. To generate signal '$O_{M,3}$' to prevent backward output-shifting faults for output 'SETUP' in "ADD PARTY/SETUP", extra signal '$O_{1,M}$' is generated as shown in a box of Figure 12. Note that only MCE can transfer signal 'O'. Here, signal '$O_{M,1}$' is synchronization signal of Tester 1, while Tester 1 applies input 'ADD PARTY' to the IUT.

The test architecture suggested in Figure 11 is more effective than conventional distributed test architecture for testing Q.2971. It is because modification of test architecture is minimal and test errors caused by channel faults becomes less while the number of call receives increases. However, there may be a bottleneck in MCE due to the increase of signal 'O's.

5. CONCLUSION AND FUTURE WORK

In this paper, a LTS generation algorithm has been proposed to test a distributed system using logical clock. Proposed algorithm classifies the events into concurrent or causal by labeling and comparing the logical clock values of the events of TS. Based on that, it generates additional signals 'C' and 'O' to control concurrent events and inserts them to the TS for each distributed object.

We have also proposed a variation of the LTS generation algorithm for which the output receive delay can be nonzero in distributed test architecture with channels among testers. It can automatically produce LTS onto already generated TS. Analyzing test scenario given with the proposed algorithm using reachability tree demonstrates that the proposed algorithm can solve the control-observation problem including output-shifting faults in a formal way.

In conventional distributed test architecture, the number of channel increases non-linearly since channels among testers are connected as a mesh. To overcome this drawback, a test architecture having O(n) channel complexity is proposed where the testers are not directly connected with each other. With this, diagnostic power of the testers can increase owing to the simplified test architecture.

As future work, over-waiting problem and appropriate ways to test distributed system using a dynamic testing method will be studied.

REFERENCES

[1] M. Benattou, L. Cacciari, R. Pasini, and O. Rafiq, "Principles and Tools for Testing Open Distributed Systems," Int'l Workshop on Testing of Communicating Systems, pp.77-92, Budapest, Hungary, September 1999.

[2] A. Ulrich and H. Konig, "Architectures for Testing Distributed Systems," Int'l Workshop on Testing of Communicating Systems, pp.93-107, Budapest, Hungary, September 1999.

[3] G. Luo, R. Dssouli, G.v. Bochmann, P. Venkataram, and A. Ghedamsi, "Test Generation With Respect To Distributed Interfaces," Computer Standards and Interfaces, pp.119-132, 1994.

[4] K. Tai, R. Carver, and E. Obaid, "Debugging Concurrent Ada Programs by Deterministic Execution," IEEE Trans. Software Engineering, Vol. 17, No. 1, pp.45-63, January 1991.

[5] M. Kim, S. T. Chanson, S. Kang, and J. Shin, "An Enhanced Model for Testing Asynchronous Communicating Systems, " FORTE/PSTV'99, June 1999.

[6] C. Fidge, "Logical Time in Distributed Computing Systems," IEEE Computer, pp.28-33, August 1991.

[7] G. Luo, G. v. Bochman, and A. Petrenko, "Test Selection Based on Communicating Nondeterministic Finite-State Machines using a Generalized Wp-Method," IEEE Trans. Software Engineering, Vol. 20, No. 2, pp.149-162, February 1994.

[8] Y. C. Young and K. C. Tai, "Observational Inaccuracy in Conformance Testing with Multiple Testers," IEEE 1st Workshop on Application-specific Software Engineering and Technology, pp.80-85, 1998.

[9] T. V. Gioles, I. Schieferdecker, M. Born, M. Winkler, and M. Li, "Configuration and Execution Support for Distributed Tests," Int'l Workshop on Testing of Communicating Systems, pp.61-76, Budapest, Hungary, September 1999.

[10] G. Coulouris, J. Dollimore, and T. Kindberg, "Distributed Systems, Concepts and Design," Second Edition, Addison-Wesley, 1994.

[11] H. Herzog and K. Sunderhaft, "General Framework for fault tolerance from ISO/ITU Reference Model for Open Distributed Processing (RM-ODP)," Object-Oriented Real-Time Dependable Systems, pp. 111-118, 1999.

[12] Y. Jung and J. Lee, "Experiences with Generation of Conformance Test Suite for Q.2971 Network-side Testing," Information Networking, pp. 286–289, 1998.

[13] ITU-T Draft Recommendation, Q.2971 B-ISDN Digital Subscriber Signaling No. 2 (DSS2) – User Network Interface layer 3 Specification for Point-to-Multipoint Call/Connection Control, 1995.

[14] Telelogic SDT3.2: Getting Started, Part1: Tutorials on SDT Tools, Telelogic, September 1997.