An Executable Protocol Test Sequence Generation Method for EFSM-specified Protocols

Chung-Ming Huang, Yuan-Chuen Lin and Ming-Yu He Jang
 Laboratory of Computer-Aided Protocol Engineering (LOCAPE)
 Institute of Information Engineering
 National Cheng Kung University
 Tainan, Taiwan 70101
 R.O.C.
 Telephone: 011-886-6-2757575 ext 62523
 Fax: 011-886-6-2747076
 E-mail: huangcm@locust.iie.ncku.edu.tw

Abstract

In this paper, we propose a method which can generate executable Extended Finite State Machine (EFSM)-based test sequences for data flow protocol test. In EFSM-specified protocols, the switch sequences can decide the executability of DO-paths. Based on the proposed Transition Executability Analysis (TEA) technique, executable switching sequences, executable DO-paths, and executable back paths, which are from the tail states of DO-paths to the initial state, can be derived. Then the complete executable data flow test sequence for an EFSM-specified protocol can be generated by concatenating the associated executable switching sequences, DO-paths, and back paths.

Keywords

Protocol Testing, Extended Finite State Machines (EFSMs), Data Flow Testing, (Executable) Test Sequence Generation.

1 Introduction

Protocol test sequence generation methods for Extended Finite State Machine (EFSM)-specified protocols are different from those for FSM-specified protocols, because of the existence of context variables and predicates [Chanson 93, Koh 94, Miller 92, Sarikaya 87, Ural 91]. One of the key issues in EFSM-based protocol test sequence generation is the executability problem. In [Chanson 93], Chanson and Zhu applied Constraint Satisfaction Problem (CSP) solving techniques, which essentially use transition self-loop analysis, to solve the test sequence executability problem. The principles of Chanson and Zhu's

*The research is supported by the National Science Council of the Republic of China under the grant NSC 84-2213-E-006-035.

Figure 1: An EFSM-specified protocol.

method is as follows: Find all paths¹ that connect variable's definitions and uses (or outputs) at the first phase. If some paths are unexecutable, then add some self-loops to make these paths executable. However, not all unexecutable transitions can find self-loops to resolve the executability problem. An example can be found in the protocol depicted in Figure 1². In Figure 1, the transition sequence (T_4, T_9, T_{11}) is an unexecutable sequence, in which variable *number* has a definition use in T_4 and has an output use in T_{11} . To have T_{11} be executable, transitions T_9 and T_{10} need to be executed twice. Thus, the final executable transition sequence from T_4 to T_{11} is $(T_4, T_9, T_{10}, T_9, T_{10}, T_9, T_{11})$. However, transitions T_9 and T_{10} are not self-loops of state *Blocked*.

In this paper, we propose a new executable test sequence generation method, which covers the data flow test of EFSM-specified protocols, using the Transition Executability Analysis (TEA) technique. Our method concatenates executable transitions during the derivation process for a test sequence. Thus, each executable test sequence, which contains a definition use of variable V and the corresponding output use of variable V , can be derived. Based on the initial status of an EFSM, in which the EFSM is in its initial state and all of the context variables are in their initial values, our method has three steps: (1) Explore an executable (switch) sequence TS_s , whose tail transition T_d contains a definition use of a variable V ; (2) derive each executable sequence TS_o rooted from the tail state of T_d , such that the tail transition T_o of TS_o contains an output use of V and there is no re-definition use of V in TS_o ; and (3) explore an executable sequence TS_b , rooted from the tail state of T_o , such that TS_b 's tail state is the initial state of the EFSM, for each TS_o .

¹For convenience, a path denotes a transition sequence in the following presentation.

²For simplicity, the initial value of each variable is 0, variables *SDU*, *no-of-segment* and *blockbound* are assigned as 2 respectively in transition T_3 . The upper bounds of *number* and *counter* are 2 and 3 respectively.

In this way, an executable test sequence can be derived by concatenating the associated TS_s , TS_o and TS_b . The complete data flow test sequence for an EFSM-specified protocol is generated by concatenating these executable test sequences.

The rest of this paper is organized as follows: In Section 2, we introduce the formal EFSM protocol specification model, and some concepts and definitions for the data flow protocol test. In Section 3, our executable test sequence generation method is presented in detail. In Section 4, we have conclusion remarks.

2 Background

In this section, we introduce the formal EFSM model that is used in our method at first. Then, some concepts and related definitions for the data flow protocol test are given.

2.1 The Formal EFSM Model

An EFSM is formally represented as a seven-tuple $\langle \Sigma, S, s_o, V, P, A, F \rangle$, where (1) Σ is the set of messages that can be sent or received, (2) S is the set of states, (3) s_o is the initial state, (4) V is the set of context variables, (5) P is the set of predicates that operate on context variables, (6) A is the set of actions that operate on context variables, (7) F is the set of state transition functions, in which each state transition function can be formally represented as follows: $S \times \Sigma \times P(V) \rightarrow S \times \Sigma \times A(V)$.

For convenience, each state transition is represents as $S_1 \xrightarrow{T} S_2$, where S_1 (S_2) is called the head (tail) state of transition T , T is called the incoming (outgoing) transition of state S_2 (S_1). Each transition contains two parts, i.e., the condition part and the action part. The condition part can contain an input event and/or a predicate. An input event is represented as "?ID.mess", in which message *mess* is input from entity *ID*. The predicate is a boolean expression that operates on context variables and parameters of the input messages. The action part can contain output events and a number of statements that operate on context variables. An output event is represented as "!ID.mess", in which message *mess* is output to entity *ID*. A transition can be fired when the condition part is satisfied. When a transition T is fired, the corresponding action part is executed, and the EFSM's state is switched from T 's head state to T 's tail state. An EFSM-specified network management (monitor) protocol is depicted in Figure 1, in which a circle represents a state and an arrow represents a transition.

Using our method, each EFSM is assumed to be a normalized EFSM. That is, there is no conditional statements, e.g., **if-then-else**, **case**, **while** loop, **for** loop, etc, in the action part of a transition. For the details of Normal Form Specifications (NFSs) for EFSMs, please refer the paper in [Sarikaya 87].

2.2 Some Concepts for the Data Flow Test

The data flow test focuses on how variables are bound to values, and how these variables are to be used. In the process of protocol testing, an Implementation Under Test (IUT) is always regarded as a black box. In order to observe how the variables are to be used, only those paths, which contain variable's definition uses and those output interactions that are influenced by these definitions, i.e., all-definition-output-paths (all-do-paths) are tested. For convenience, the following definitions are used in the protocol test sequence generation that follows the all-do-paths criterion.

Definition 1: (state configuration)

A state (context variables) configuration $S_i(V_{i1}, V_{i2}, \dots, V_{in})$, which is abbreviated as S_i ,

Set Transition	A-use	I-use	P-use	O-use	Define Element (variable, transition_id)	Executable Switching Sequence (ESS)
T1						
T2						
T3	number counter	SDU no. of segment blockbound			(number, 3) (counter, 3) (SDU, 3) (no. of segment, 3)	(T1, T2, T3) (T1, T2, T3) (T1, T2, T3) (T1, T2, T3)
T4	number			SDU number	(blockbound, 3)	(T1, T2, T3)
T5					(number, 4)	(T1, T2, T3, T4)
T6					(number, 4)	(T1, T2, T3, T4, T6, T4)
T7			number no. of segment	counter	(number, 4) (number, 4)	(T1, T2, T3, T4, T9, T12, T4) (T1, T2, T3, T4, T9, T10, T9, T12, T4)
T8	number		number no. of segment	SDU number	(number, 8) (number, 8)	(T1, T2, T3, T4, T8) (T1, T2, T3, T4, T9, T10, T8)
T9	counter				(number, 8) (counter, 9)	(T1, T2, T3, T4, T9, T10, T9, T10, T8) (T1, T2, T3, T4, T9)
T10			counter blockbound		(counter, 9)	(T1, T2, T3, T4, T8, T9)
T11			counter blockbound	number	(counter, 9)	(T1, T2, T3, T4, T9, T10, T9)
T12			counter blockbound		(counter, 9)	(T1, T2, T3, T4, T8, T9, T10, T9)
T13					(counter, 9)	(T1, T2, T3, T4, T9, T10, T9, T10, T9)
T14						
T15						
T16						
T17						

(a)

(b)

Table 1: (a) The sets of A-use, I-use, P-use and O-use in each transition; (b) the ESSs for each A-use and I-use.

of an EFSM is a snapshot of the EFSM such that the EFSM is at state S_i , and the values of variables $V_1, V_2, \dots$, and V_n of the EFSM are $V_{i1}, V_{i2}, \dots, V_{in}$ respectively.

Definition 2: (reachable state configuration)

A state configuration $S_i(V_{i1}, V_{i2}, \dots, V_{in})$ is a reachable state configuration if there is an executable transition path $(t_1, t_2, \dots, t_m)$, such that transition t_1 's head state configuration is the initial state configuration and $S_i(V_{i1}, V_{i2}, \dots, V_{in})$ is transition t_m 's tail state configuration.

Definition 3: (A-use, I-use, P-use, O-use)

The use of a variable x is an assignment-use (*A-use*) in a transition if x appears at the left hand side of an assignment statement of the transition. When a variable x appears in the input portion of a transition, the use of x is an input-use (*I-use*) in the transition. When a variable x appears in the predicate expression of a transition, the use of x is a predicate-use (*P-use*) in the transition. When a variable x appears in an output statement of a transition, the use of x is an output-use (*O-use*) in the transition.

Both *A-use* and *I-use* are called definition-use (*D-use*). Table 1-(a) shows the uses of all variables and the associated transition identifiers in the protocol depicted in Figure 1.

Definition 4: (global use, local use)

The use of a variable x in a transition is a *global use* when x 's *D-use* occurs in some other transitions, otherwise the use of x is a *local use*.

Definition 5: (global def, local def)

The definition of a variable x in a transition is a *global def* if there is a global use of x in some other transitions, otherwise the definition of x is a *local def*.

Definition 6: (def-clear path)

A path $(t_1, \dots, t_n)$ is a *def-clear path* with respect to a variable x if transitions $t_2, \dots, t_{n-1}$ do not contain any *D-use* of x .

Definition 7: (DO-path)

A path $(t_1, \dots, t_n)$ is a *DO-path* with respect to a variable x if t_1 has a *global def* of x and

Figure 2: (a) Association between ESSs and DO-paths; (b) the TEA tree used for detecting the ESSs with respect to the A-use of variable *number* in transition T_3 .

either:

- (1) t_n has an O-use of x and the path is *def-clear* path with respect to x , or
- (2) t_k contain a *P-use* of x , where $2 \leq k \leq n - 1$, the path $t_1, \dots, t_k, \dots, t_{n-1}$ is a *def-clear* path with respect to x , such that (i) t_n has an O-use of any variable, and (ii) the execution of t_n depends on the execution of t_k , i.e., t_n indirectly depends on the predicate of t_k .

Definition 8: (IS-connected)

An EFSM M is *IS-connected*, where IS represents the initial state of M , if each reachable state configuration S has at least one executable path T , such that S is T 's head state and IS in T 's tail state.

Definition 9: (bounded)

An EFSM M is bounded, if for each variable x in M , x 's legal values is bounded in a finite set of $\text{bound}(x)$, i.e., the number of x 's legal values is finite.

Definition 10: (TEA tree)

The Transition Executability Analysis (TEA) of an EFSM M generates an expanding tree rooted from a given state configuration of M , such that (1) each circle in the tree represents a reachable state configuration, (2) each arc in the tree represents an executable transition originated from the corresponding head state configuration.

3 Test Sequence Generation for the EFSM Model

In this section, we present our executable test sequences generation method for data flow test. For an EFSM-specified protocol M , such that M is (1) IS-connected and (2) bounded, the generated executable test sequences (ETs) cover all of the executable DO-paths which exist in M using our method.

Our method contains three main steps. In the first step, executable switch sequences (ESSs), which can reach DO-paths, are generated by expanding a TEA tree rooted from

the EFSM's initial state configuration. In the second step, all of the executable DO-paths (EDO-paths), which are generated by expanding TEA trees rooted from the state configurations of the tail states of ESSs, are selected. When all of the EDO-paths have been selected, the third step is invoked. In the third step, an executable back path from each EDO-path's tail state to the initial state of the EFSM is explored. An executable back path for an EDO-path P is derived by expanding a TEA tree rooted from the tail state of P.

Before presenting the derivation procedures in detail, some notations are introduced to simplify the presentation. Let A and B be two transition sequences. Then, the concatenation C of A and B is represented as $C = A @ B$. TSP1 and TSP2 are temporary state pools storing existing state configurations. IS is the initial state. DE, which identifies the definition use, i.e., A-use or I-use, of a variable in a transition, contains two attributes: (1) Var: the variable identifier, and (2) Tid: the identifier of the transition which contains the definition use of the associated variable Var. A-use(T), I-use(T), P-use(T) and O-use(T) are sets, which identify variables that belong to A-use, I-use, P-use and O-use in transition T, respectively. Each element in A-use(T), I-use(T), P-use(T) and O-use(T) has the similar attributes as that of DE. SE, which denotes a state element in TSP_i, $i=1$ or 2, contains three attributes: (1) State: the state identifier and the associated context variables' values, i.e., the state configuration, (2) Level: the level (number) of the associated state configuration (of the SE) in the TEA tree, in which the level of the root state configuration is 1, (3) Path: the path from the root state configuration to the associated state configuration (of the SE) in the TEA tree. Function $\text{tailstate}(TT)$ returns the tail state configuration of the transition sequence TT. Function $\text{exist}(P, PP)$ returns a path TP from path pool PP such that path P is a postfix sequence of TP; otherwise, an infinitely long sequence, which is represented by the symbol ∞ , is returned. Function $\text{shorter}(P, T)$ returns the shorter path of path P and path T.

3.1 The Derivation of Executable Switch Sequences (ESSs)

In order to test DO-paths of an EFSM M with respect to a variable x which has either A-use or I-use in a transition T_i , executable switch sequences which are originated from the initial state of M to the tail state of T_i need to be detected. For an A-use or I-use of a variable x in a transition, there may be more than one DO-path and there may be more than one ESS. The key issue is that these DO-paths may need different ESSs to be executable. An example is given in Figure 2-(a). In Figure 2-(a), S_i is the initial state, an A-use of x is in T_4 . In order to test the data flow with respect to x in T_4 , an ESS must be applied so that T_4 can be executable. In Figure 2, there are two ESSs which can reach T_4 , one is $(T_1, \dots, T_2, T_4)$ and the other one is $(T_1, \dots, T_2, T_3, T_4)$ if the bound of variable *counter* is restricted from 1 to 2. There are two DO-paths with respect to x in T_4 , one is $(T_4, \dots, T_5, T_6, \dots, T_8)$ and the other one is $(T_4, \dots, T_5, T_7, \dots, T_9)$. The DO-path $(T_4, \dots, T_5, T_6, \dots, T_8)$ can be executable only when the ESS $(T_1, \dots, T_2, T_4)$ is selected, and the DO-path $(T_4, \dots, T_5, T_7, \dots, T_9)$ can be executable only when the ESS $(T_1, \dots, T_2, T_3, T_4)$ is selected. If only one of the above two ESSs is used to search for these two DO-paths, one of the two DO-paths will be lost. Thus, for an A-use or I-use of a variable x in a transition T_i , all of the possible ESSs, which result in different T_i 's head state configurations, need to be detected. The following procedure generates all of the ESSs for a DE.

Procedure ESS(IS, DE)

 root.State := IS

 root.Level := 1

 root.Path := $\emptyset$

 TSP1, TSP2 := $\emptyset$

 ESS-pool := $\emptyset$

 add root to TSP1 and TSP2 /* TSP2 is used for duplication check */

```

L := 1
while TSP1 is not empty do
  while TSP1 contains elements whose Levels are L do
    remove an SE from TSP1 such that SE.Level = L
    for each executable outgoing transition  $T_i$  of SE, such that (tailstate( $T_i$ ) is not in TSP2
      and all variables' values in tailstate( $T_i$ ) are in their bounds) or (tailstate( $T_i$ ) is a
      duplication of a state S that is stored in TSP2 and none of S's incoming transitions
      is  $T_i$ ) do
      if  $DE \in A\text{-use}(T_i)$  or  $DE \in I\text{-use}(T_i)$ 
      then
        ESS := SE.Path @  $T_i$ 
        add the ESS to ESS-pool
      endif
      tailstate( $T_i$ ).Level := L+1
      tailstate( $T_i$ ).Path := SE.Path @  $T_i$ 
      add tailstate( $T_i$ ) to TSP1 and TSP2
    endfor
  endwhile
  L := L+1
endwhile /* end of procedure */

```

The procedure of ESS(IS, DE) searches for the ESSs of the states whose outgoing transitions contain some definition uses of some variables. Initially, the ESS(IS, DE) procedure sets the initial state IS as the root of the TEA tree. Then, expand the TEA tree one state by one state and one level by one level, i.e., expand the TEA tree using the Bread-First-Search (BFS) way, until the TSP1 pool becomes empty, i.e., all of the possible ESSs have been explored for the definition use of DE.Var in DE.Tid. The *for* loop explores all of executable outgoing transitions of a state element. An outgoing transition T_i is checked whether (1) T_i 's tail state configuration doesn't exist in the TEA tree, where TSP2 stores all of currently existing state configurations in the TEA tree, and all of the variables' values are in their bounded values after T_i having been executed, or (2) T_i 's tail state configuration S has been reached previously in the TEA tree, but the previous S is not generated by executing T_i . If the answer is negative, then stop expanding this path; otherwise check whether $A\text{-use}(T_i)$ or $I\text{-use}(T_i)$ contains the DE element. If DE is not contained in $A\text{-use}(T_i)$ or $I\text{-use}(T_i)$, then continue to expand the TEA tree; otherwise, the path from root to T_i 's head state is an executable switching sequence to be obtained. For a transition T_i that has an A-use or I-use of variable x, and a transition T_o that has an O-use of variable x, there may be two executable switching sequences, ESS_i and ESS_o , such that (1) T_i is the tail transition of both ESS_i and ESS_o , (2) the length of ESS_i is greater than that of ESS_o , (3) the tail state configuration of ESS_i is different from that of ESS_o , and (4) the length from ESS_i to T_o is shorter than that from ESS_o to T_o . That is, a longer executable switching sequence may result in a shorter executable DO-path. Thus, the tail state of T_i , which contains an A-use or I-use of a variable x, still needs to be added in TSP1 to explore the other possible executable switching sequences. Repeat the above procedure, until all of the ESSs have been detected. Figure 2-(b) shows the TEA tree used for detecting the ESS with respect to the A-use of variable *number* in transition T_3 . Table 1-(b) shows all of the ESSs for each A-use or I-use of the protocol depicted in Figure 1.

Theorem 1:

The TEA tree's state space for the ESS(IS, DE) procedure is finite. That is, the ESS(IS, DE) procedure can be terminated in finite steps.

Proof:

Since all variables' values are bounded, the number of possibly generated state configurations is $GSC = |S| * |V_1| * \dots * |V_n|$, where $|S|$ is the number of states, and $|V_i|$ is the number

of legal values of variable V_i , $i=1, \dots, n$. Let $k = \text{maximum}(\text{out}(S_1), \text{out}(S_2), \dots, \text{out}(S_{|S|}))$, where $\text{out}(S_i)$ is the number of state S_i 's outgoing transitions and $i=1, \dots, |S|$. Hence, the state space of the TEA tree is $GSC * k$. In other words, the state space of the TEA tree is finite, i.e., the expanding of a TEA tree can be terminated in finite steps.

3.2 The Derivation of Executable DO-paths (EDO-paths)

For an ESS, which contains an A-use or I-use of a variable x in ESS's tail transition T_i , all of the DO-paths with respect to x need to be checked based on ESS's tail state configuration. The DO-paths to be obtained have two types: (1) Let T_{i+n} have an O-use of x , and $T_i, \dots, T_{i+n}$ be a def-clear path with respect to x and be executable. The path $(T_i, \dots, T_{i+n})$ is a DO-path of type 1. In this condition, x can be directly observed using the output event in T_{i+n} . (2) Let T_{i+n} have a P-use of x and $T_i, \dots, T_{i+n}$ be a def-clear path with respect to x and be executable, and a nearest O-use be in T_{i+n+k} such that the path $(T_i, \dots, T_{i+n}, \dots, T_{i+n+k})$ be executable. The path $(T_i, \dots, T_{i+n}, \dots, T_{i+n+k})$ is a DO-path of type 2. In this condition, the effect of the P-use of x can be observed indirectly by observing the output event of T_{i+n+k} .

Procedure EDO-path-search (ESS, x)

root.State := tailstate(ESS)

root.Level := 1

root.Path := $\emptyset$

TSP1, TSP2 := $\emptyset$

add root to TSP1 and TSP2 /* TSP2 is used for duplication check */

L := 1

while TSP1 is not empty **do**

while TSP1 contains elements whose Levels are L **do**

 remove an SE from TSP1, such that SE.Level = L

for each executable outgoing transition T_i of SE, such that all variables' values in tailstate(T_i) are in their bounds **do**

Case 1: $x \in \text{O-use}(T_i)$ and $x \notin \text{I-use}(T_i)$ and $x \notin \text{A-use}(T_i)$

 DO-path := SE.Path @ T_i

 EDO-path := shorter(ESS @ DO-path, exist(DO-path, EDO-path-pool))

 add EDO-path to EDO-path-pool

 expand-or-discard(tailstate(T_i))

Case 2: $x \in \text{O-use}(T_i)$ and $x \notin \text{I-use}(T_i)$ and $x \in \text{A-use}(T_i)$ and (the execution of the statement containing O-use of x is before that of the statement containing A-use of x)

 DO-path := SE.Path @ T_i

 EDO-path := shorter(ESS @ DO-path, exist(DO-path, EDO-path-pool))

 add EDO-path to EDO-path-pool

Case 3: $x \in \text{P-use}(T_i)$ and $x \notin \text{I-use}(T_i)$

 tailstate(T_i).Path := SE.Path @ T_i

if O-use(T_i) is not empty

then

 DPO-path = tailstate(T_i).Path

 EDO-path := shorter(ESS @ DPO-path, exist(DPO-path, EDO-path-pool))

 add EDO-path to EDO-path-pool

else

 DPO-path-search(tailstate(T_i))

endif

if $x \notin \text{A-use}(T_i)$

then

 expand-or-discard(tailstate(T_i))

endif


```

    Case 4:  $x \notin \text{I-use}(T_i)$  and  $x \notin \text{P-use}(T_i)$  and  $x \notin \text{A-use}(T_i)$  and  $x \notin \text{O-use}(T_i)$ 
        expand-or-discard(tailstate( $T_i$ ))
    Case 5: otherwise, stop expanding this path
endfor
endwhile
L := L+1
endwhile /* end of procedure */

Procedure expand-or-discard(S)
if S is not a duplication of any state configuration in TSP2
then
    S.Path := SE.Path @  $T_i$ 
    S.Level := L+1
    add S to TSP1 and TSP2
endif /* end of procedure */

Procedure DPO-path-search(S)
root.State := S
root.Level := 1
TSP :=  $\emptyset$ 
add root to TSP
L := 1
while TSP is not empty do
    while TSP contains elements whose Levels are L do
        remove an SE from TSP, such that SE.Level = L
        for each executable outgoing transition  $T_j$  of SE do
            if O-use( $T_j$ ) is not empty
            then
                DPO-path := SE.Path @  $T_j$ 
                EDO-path := shorter(ESS @ DPO-path, exist(DPO-path, EDO-path-pool))
                add EDO-path to EDO-path-pool
            exit
            else
                tailstate( $T_j$ ).Path := SE.Path @  $T_j$ 
                add tailstate( $T_j$ ) to TSP
            endif
        endfor
    endwhile
    L := L+1
endwhile /* end of procedure */

```

For a given variable x , and an ESS in which ESS's tail transition has a definition use, i.e., A-use or I-use, of x , the procedure of EDO-path-search(ESS, x) searches for the EDO-paths using the Bread-First-Search (BFS) way. Initially, the tail state of the ESS is set as the root of the TEA tree. All of the state elements in the TEA tree is explored one state by one state and one level by one level. Each executable outgoing transition T_i , in which all of the variables' values are in their bounds after executing T_i , of a selected state configuration in TSP1 is checked. The use of variable x in a transition T_i can be classified into five cases:

- Case 1: T_i has no re-definition use but has an O-use of x . In this case, the path which is from ESS's tail state to the tail state of T_i is a DO-path with respect to x . The DO-path is appended with the ESS to form an EDO-path. However, there may exist some EDO-paths which have the same DO-path but with different ESSs. A shorter EDO-path with respect to the same DO-path should be selected.

Functions *exist* and *shorter* are called to have the shorter EDO-path with respect to the DO-path. When an EDO-path $(T_1, \dots, T_i, \dots, T_{k_1})$, which contains a D-use in T_i and an O-use (or P-use) in T_{k_1} of a variable V , is detected, it is still possible that there is an EDO-path $(T_1, \dots, T_i, \dots, T_{k_1}, \dots, T_{k_2})$, such that T_{k_2} has an O-use (or P-use) of V . Therefore, the tail state configuration of a newly detected EDO-path still needs to have further exploration. The *expand-or-discard*(*tailstate*(T_i)) procedure is called to decide whether continue to expand the path or not. The *expand-or-discard*(*tailstate*(T_i)) procedure checks whether the tail state S of a path P is a duplication or not. If the answer is positive, then stop expanding path P ; otherwise, add S to TSP1 and TSP2 to further expand path P .

- Case 2: T_i has no I-use of x , but has an A-use and an O-use of x , and the execution of the statement containing O-use of x is before that of the statement containing A-use of x in T_i . In this case, the DO-path is valid. Thus, the corresponding actions are the same as that of Case 1, except the expanding of the path is stop at the tail state of T_i , because there is a re-definition use of x in T_i .
- Case 3: T_i has no I-use but has a P-use of x . In this case, if T_i has an output event, then the path which is from ESS's tail state to the tail state of T_i is a DO-path of type 2 with respect to x ; otherwise, the *DPO-path-search*(*tailstate*(T_i)) procedure is called to search for the DO-path of type 2 with respect to x . In the procedure of the *DPO-path-search*(*tailstate*(T_i)), a new TEA tree is expanded until a transition T_j containing an output event is reached³. When the DO-path is obtained, the DO-path is appended with the ESS to form the EDO-path. Functions *exist* and *shorter* are called to have the shorter EDO-path with respect to the DO-path. Then, if T_i does not have an A-use of x , the *expand-or-discard*(*tailstate*(T_i)) procedure is called to decide whether continue to expand the path or not; otherwise, if T_i has an A-use of x , stop expanding this path.
- Case 4: variable x is not used in T_i . In this case, the *expand-or-discard*(*tailstate*(T_i)) procedure is called to decide whether continue to expand this path or not.
- Case 5: The use of variable x is not in above cases, i.e., there is a re-definition use in T_i , stop expanding this path.

Repeat the above steps, until all EDO-paths with respect to x have been detected. Figure 3 shows the TEA tree used for detecting DO-paths with respect to the A-use of variable *number* in transition T_3 . Table 2 shows all of the EDO-paths of the protocol depicted in Figure 1.

Theorem 2:

The TEA tree's state space for *EDO-path-search*(ESS, x) is finite. That is, the *EDO-path-search*(ESS, x) procedure can be terminated in finite steps.

Proof:

The proof of this theorem is the same as that for Theorem 1.

3.3 The Derivation of Executable Back Paths

When an EDO-path is derived, the back path from the tail state of the EDO-path to the initial state IS needs to be generated, such that IUT can be brought to the initial state to have the next test sequence. The following EBP-path-search procedure can search for the back path for an EDO-path.

³Since the EFSM is IS-connected, an output event can be detected eventually.

```

Procedure EBP-path-search(EDO)
root.State := tailstate(EDO)
root.Level := 1
root.Path := EDO
TSP :=  $\emptyset$ 
add root to TSP
L := 1
while TSP is not empty do
  while TSP contains elements whose Levels are L do
    remove an SE from TSP, such that SE.Level = L
    for each executable outgoing transition  $T_i$  of SE do
      if tailstate( $T_i$ ) = IS
        then
          EBP-path := SE.Path @  $T_i$ 
          add EBP-path to EBP-path-pool
          exit
        else
          if tailstate( $T_i$ ) is not in TSP
            then
              tailstate( $T_i$ ).Path := SE.Path @  $T_i$ 
              add tailstate( $T_i$ ) to TSP
            endif
          endif
        endfor
      endwhile
      L := L+1
    endwhile /* end of procedure */

```

Starting from the tail state of an EDO-path, the procedure of EBP-path-search(EDO) expands one state by one state and one level by one level. If the initial state is not reached at level L, level L+1 is explored. The procedure is continued until the initial state of the EFSM is reached⁴. In this way, a complete executable test sequence with respect to a DO-path is derived.

Theorem 3:

The TEA tree's state space for the EBP-path-search(EDO) procedure is finite. That is, the EBP-path-search(EDO) procedure can be terminated in finite steps.

Proof:

Since the EFSM is *IS-connected*, there is an EBP-path for each EDO-path according to **Definition 7**. Using the same calculation as that in the *proof* for **Theorem 1** (and 2), the EBP-path can be detected in finite steps.

3.4 Generation of Executable Test Sequences (GETS)

Using the ESS(IS, DE), EDO-path-search(ESS, x), and EBP-path-search(EDO) procedures, all of the executable test sequences covering all of the executable DO-paths can be generated. The executable test sequence generation algorithm is as follows:

Algorithm Generating-Executable-Test-Sequence (GETS)

⁴For simplicity, it is assumed that all variable values can be reset to their initial values respectively. It can be achieved as follows. For each $S \xrightarrow{T} S_{init}$, in which S_{init} is the initial state and T is an incoming transition of S_{init} , a split state S' and a reset transition T_r are added, such that $S \xrightarrow{T} S' \xrightarrow{T_r} S_{init}$. Each variable is reset to its initial value in T_r . Otherwise, the EBP-path-search procedure keeps expanding until an ESS's head state configuration is reached.

Define-Element (variable, transition_id)	ESS	TYPE (DO/DPO)	(EDO_PATH; #PATH_ID)
(number, 3)	T1,T2,T3	DO	(ESS,T4 ; #1), (ESS,T4,T6,T4 ; #2), (ESS,T4,T8 ; #3), (ESS,T4,T8,T9,T10,T9,T10,T9,T11 ; #4), (ESS,T4,T9,T12,T4 ; #5), (ESS,T4,T9,T10,T8 ; #6), (ESS,T4,T9,T10,T9,T12,T4 ; #7), (ESS,T4,T9,T10,T9,T10,T9,T11 ; #8).
		DPO	(ESS,T4,T8,T9,T10,T7 ; #9), (ESS,T4,T8,T9,T10,T9,T10,T7 ; #10), (ESS,T4,T8,T7 ; #11).
		DO	(ESS,T4,T8,T7 ; #12).
(counter, 3)	T1,T2,T3	DPO	(ESS,T4,T8,T9,T10,T6 ; #13), (ESS,T4,T8,T9,T10,T7 ; #14), (ESS,T4,T8,T9,T10,T9,T10,T6 ; #15), (ESS,T4,T8,T9,T10,T9,T10,T7 ; #16), (ESS,T4,T8,T9,T10,T9,T10,T9,T11 ; #17), (ESS,T4,T8,T9,T10,T9,T12 ; #18), (ESS,T4,T9,T10,T6 ; #19), (ESS,T4,T9,T10,T9,T10,T7 ; #20), (ESS,T4,T9,T10,T9,T10,T9,T11 ; #21), (ESS,T4,T9,T10,T9,T12 ; #22), (ESS,T4,T9,T12 ; #23), (ESS,T4,T9,T10,T9,T10,T6 ; #24), (ESS,T4,T9,T10,T9,T10,T8 ; #25).
(SDU, 3)	T1,T2,T3	DO	(ESS,T4 ; #26), (ESS,T4,T6,T4 ; #27), (ESS,T4,T8 ; #28), (ESS,T4,T9,T12,T4 ; #29), (ESS,T4,T9,T10,T8 ; #30), (ESS,T4,T9,T10,T9,T10,T8 ; #31), (ESS,T4,T9,T10,T9,T12,T4 ; #32).
(no_of_segment, 3)	T1,T2,T3	DPO	(ESS,T4,T8 ; #33), (ESS,T4,T8,T9,T10,T7 ; #34), (ESS,T4,T8,T9,T10,T9,T10,T7 ; #35), (ESS,T4,T8,T7 ; #36).
(blockbound, 3)	T1,T2,T3	DPO	(ESS,T4,T8,T9,T10,T6 ; #37), (ESS,T4,T8,T9,T10,T7 ; #38), (ESS,T4,T8,T9,T10,T9,T10,T6 ; #39), (ESS,T4,T8,T9,T10,T9,T10,T7 ; #40), (ESS,T4,T8,T9,T10,T9,T10,T9,T11 ; #41), (ESS,T4,T8,T9,T10,T9,T12 ; #42), (ESS,T4,T9,T10,T9,T12 ; #43), (ESS,T4,T9,T10,T8 ; #44), (ESS,T4,T9,T10,T6 ; #45), (ESS,T4,T9,T10,T9,T10,T6 ; #46), (ESS,T4,T9,T10,T9,T10,T8 ; #47), (ESS,T4,T9,T10,T9,T10,T9,T11 ; #48), (ESS,T4,T9,T12 ; #49).
(number, 4)	T1,T2,T3, T4	DO	(ESS,T6,T4 ; #50), (ESS,T8 ; #51), (ESS,T8,T9,T10,T9,T10,T9,T11 ; #52), (ESS,T9,T12,T4 ; #53), (ESS,T9,T10,T8 ; #54), (ESS,T9,T10,T9,T12,T4 ; #55), (ESS,T9,T10,T9,T10,T9,T11 ; #56).
		DPO	(ESS,T8,T9,T10,T7 ; #57), (ESS,T8,T9,T10,T9,T10,T7 ; #58), (ESS,T8,T7 ; #59).
	T1,T2,T3, T4,T6,T4	DO	(ESS,T9,T10,T9,T10,T9,T11 ; #60).
		DPO	(ESS,T7 ; #61), (ESS,T9,T10,T7 ; #62), (ESS,T9,T10,T9,T10,T7 ; #63).
	T1,T2,T3, T4,T9,T12, T4	DO	(ESS,T9,T10,T9,T11 ; #64).
		DPO	(ESS,T7 ; #65), (ESS,T9,T10,T7 ; #66).
	T1,T2,T3, T4,T9,T10, T9,T12,T4	DO	(ESS,T9,T11 ; #67).
		DPO	(ESS,T7 ; #68).
	T1,T2,T3, T4,T8	DO	(ESS,T9,T10,T9,T10,T9,T11 ; #69).
		DPO	(ESS,T7 ; #70), (ESS,T9,T10,T9,T10,T7 ; #72), (ESS,T9,T10,T9,T11 ; #73).
(number, 8)	T1,T2,T3,T4, T9,T10,T8	DO	(ESS,T7 ; #74), (ESS,T9,T10,T7 ; #75).
	T1,T2,T3,T4, T9,T10,T9, T10,T8	DO	(ESS,T9,T11 ; #76).
		DPO	(ESS,T7 ; #77).
	T1,T2,T3,T4, T8,T9	DPO	(ESS,T10,T6 ; #78), (ESS,T10,T7 ; #79), (ESS,T10,T9,T10,T7 ; #80), (ESS,T10,T9,T10,T6 ; #81), (ESS,T10,T9,T10,T9,T11 ; #82).
		DO	(ESS,T12,T4,T7 ; #83), (ESS,T10,T9,T12,T4,T7 ; #84).
(counter, 9)	T1,T2,T3, T4,T9	DPO	(ESS,T12 ; #85), (ESS,T12,T4,T9,T10,T6 ; #86), (ESS,T12,T4,T9,T10,T7 ; #87), (ESS,T12,T4,T9,T10,T9,T11 ; #88), (ESS,T10,T6 ; #89), (ESS,T10,T8 ; #90), (ESS,T10,T9,T12 ; #91), (ESS,T10,T9,T12,T4,T9,T11 ; #92), (ESS,T10,T9,T10,T6 ; #93), (ESS,T10,T9,T10,T8 ; #94), (ESS,T10,T9,T10,T9,T11 ; #95).
			(ESS,T10,T8,T7 ; #96), (ESS,T12,T4,T7 ; #97).
			(ESS,T10,T6 ; #98), (ESS,T10,T8 ; #99), (ESS,T10,T8,T9,T11 ; #100), (ESS,T10,T9,T11 ; #101), (ESS,T12,T4,T9,T11 ; #102).
			(ESS,T10,T6 ; #103), (ESS,T10,T8 ; #104), (ESS,T10,T9,T11 ; #105).
			(ESS,T11 ; #106).
	T1,T2,T3,T4, T8,T9,T10,T9, T10,T9	DPO	(ESS,T11 ; #107).
	T1,T2,T3,T4, T9,T10,T9, T10,T9	DPO	
	T1,T2,T3,T4, T8,T9,T10,T9, T9,T10,T9, T10,T9	DPO	
	T1,T2,T3,T4, T8,T9,T10, T9,T10,T9	DPO	
	T1,T2,T3,T4, T8,T9,T10, T9,T10,T9	DPO	

Table 2: The EDO-paths for the protocol depicted in Figure 1.

NOTE: In the field of "Test Target", (variable, [PATH_ID]) represents the PATH_IDs and the corresponding defined variable that are contained in the associated ETS.

ETS No.	Executable Test Sequence (ETS)	Test Target (variable [PATH_ID])
1	(T1.T2.T3.T4.T6.T4.T7.T16)	(number, [1,2,50,61]), (SDU, [26,27])
2	(T1.T2.T3.T4.T6.T4.T9.T10.T7.T16)	(number, [1,2,50,62]), (SDU, [26,27])
3	(T1.T2.T3.T4.T6.T4.T9.T10.T9.T10.T7.T16)	(number, [1,2,50,63]), (SDU, [26,27])
4	(T1.T2.T3.T4.T6.T4.T9.T10.T9.T10.T9.T11.T16)	(number, [1,2,50,60]), (SDU, [26,27])
5	(T1.T2.T3.T4.T8.T9.T10.T7.T16)	(number, [1,3,9,71]), (counter, [14,79]), (SDU, [26,28]), (no_of_segment, [33,34]), (blockbound, [38])
6	(T1.T2.T3.T4.T8.T9.T10.T9.T10.T7.T16)	(number, [1,3,10,51,58,72]), (counter, [16,80,104]), (SDU, [26,28]), (no_of_segment, [33,35]), (blockbound, [40])
7	(T1.T2.T3.T4.T8.T9.T10.T9.T10.T9.T11.T16)	(number, [1,3,4,51,52,69]), (counter, [17,82,105,107]), (SDU, [26,28]), (no_of_segment, [33]), (blockbound, [41])
8	(T1.T2.T3.T4.T8.T7.T16)	(number, [1,3,11,51,59,70]), (counter, [12]), (SDU, [26,28]), (no_of_segment, [33,36])
9	(T1.T2.T3.T4.T9.T10.T9.T10.T9.T11.T16)	(number, [1,8,56]), (counter, [21,95,101,106]), (SDU, [26]), (blockbound, [48])
10	(T1.T2.T3.T4.T8.T9.T10.T6.T4.T9.T10.T9.T11.T16)	(number, [1,3]), (counter, [13,78]), (SDU, [26,28]), (no_of_segment, [33]), (blockbound, [37])
11	(T1.T2.T3.T4.T8.T9.T10.T9.T10.T6.T4.T9.T11.T16)	(number, [1,3,51]), (counter, [15,81,103]), (SDU, [26,28]), (no_of_segment, [33]), (blockbound, [39])
12	(T1.T2.T3.T4.T8.T9.T10.T9.T12.T4.T9.T11.T16)	(number, [1,3,51]), (counter, [18]), (SDU, [26,28]), (no_of_segment, [33]), (blockbound, [42])
13	(T1.T2.T3.T4.T9.T10.T6.T4.T7.T16)	(number, [1]), (counter, [19,89]), (SDU, [26]), (blockbound, [45])
14	(T1.T2.T3.T4.T9.T10.T9.T10.T6.T4.T7.T16)	(number, [1]), (counter, [24,93,98]), (SDU, [26]), (blockbound, [46])
15	(T1.T2.T3.T4.T9.T10.T8.T9.T10.T7.T16)	(number, [1,6,54,75]), (counter, [20,90]), (SDU, [26,30]), (blockbound, [44])
16	(T1.T2.T3.T4.T9.T10.T8.T9.T10.T9.T11.T16)	(number, [1,6,54,73]), (counter, [20,90]), (SDU, [26,30]), (blockbound, [44])
17	(T1.T2.T3.T4.T9.T10.T9.T10.T8.T7.T16)	(number, [1,77]), (counter, [25,94,96,99]), (SDU, [26,31]), (blockbound, [47])
18	(T1.T2.T3.T4.T9.T10.T9.T10.T8.T9.T11.T16)	(number, [1,76]), (counter, [25,94,100]), (SDU, [26,31]), (blockbound, [47])
19	(T1.T2.T3.T4.T9.T12.T4.T7.T16)	(number, [1,5,53,65]), (counter, [23,85]), (SDU, [26,29]), (blockbound, [49])
20	(T1.T2.T3.T4.T9.T12.T4.T9.T10.T6.T4.T9.T11.T16)	(number, [1,5,53]), (counter, [23,85,86]), (SDU, [26,29]), (blockbound, [49])
21	(T1.T2.T3.T4.T9.T12.T4.T9.T10.T7.T16)	(number, [1,5,53,66]), (counter, [23,85,87]), (SDU, [26,29]), (blockbound, [49])
22	(T1.T2.T3.T4.T9.T12.T4.T9.T10.T9.T11.T16)	(number, [1,5,53,64]), (counter, [23,85,88]), (SDU, [26,29]), (blockbound, [49])
23	(T1.T2.T3.T4.T9.T10.T9.T12.T4.T7.T16)	(number, [1,7,55,68]), (counter, [22,102]), (SDU, [26,32]), (blockbound, [43])
24	(T1.T2.T3.T4.T9.T10.T9.T12.T4.T9.T11.T16)	(number, [1,7,55,67]), (counter, [22,92,102]), (SDU, [26,32]), (blockbound, [43])

Table 3: Final ETSS for the protocol depicted in Figure 1.

```

Step 1:
  D-pool :=  $\emptyset$ , EDO-path-pool :=  $\emptyset$ 
  for each transition T do
    generate the sets of A-use(T), I-use(T), P-use(T), and O-use(T)
    add A-use(T) and I-use(T) to D-pool
  endfor
Step 2:
  remove an element DE from D-pool
  ESS(IS, DE) /* in Section 3.1 */
Step 3:
  for each sequence ESS in ESS-pool do
    remove an ESS from ESS-pool
    EDO-path-search(ESS, DE.variable) /* in Section 3.2 */
  endfor
Step 4:
  If D-pool is not empty, go to Step 2; otherwise, go to Step 5.
Step 5:
  for each sequence EDO in EDO-path-pool do
    EBP-path-search(EDO) /* in Section 3.3 */
  endfor

```

The Generating-Executable-Test-Sequence (GETS) algorithm is explained as follows: In Step 1, the four sets of A-use(T), I-use(T), P-use(T) and O-use(T) for each transition T are generated. Additionally, all of the A-use and I-use elements are added to D-pool in order to find the DO-paths. In Step 2, each DE element in the D-pool is explored. ESSs for each DE element are derived using the ESS(IS, DE) procedure. In Step 3, all of the shortest EDO-paths with respect to a DE element are generated. In Step 4, repeat Step 2 and Step 3 until all of the EDO-paths for all DE elements have been detected. Step 5 searches for the back path from the tail state of each EDO-path to the initial state IS. Appending the back path to an EDO-path forms an executable test sequence. Table 3 shows all of the executable test sequences for the protocol depicted in figure 1.

4 Conclusion

In this paper, we have presented a new data flow protocol test sequence generation method for EFSM-specified protocols. An executable test sequence (ETS) contains three paths: (1) the executable switch sequence (ESS); (2) the executable DO-path (EDO-path); and (3) the executable back path (EBP-path). For solving the executability problem, we have proposed the Transition Executability Analysis (TEA) technique to analyze paths' executability. Using the TEA technique, which is expanded in the Bread-First-Search (BFS) way, (1) each shortest executable switch sequence, which connects the initial state to the head state of the transition containing an A-use or I-use of a variable x, is derived based on the initial state's configuration; (2) the executable DO-paths (EDO-paths) with respect to a variable x are derived based on the associated ESSs' tail state configurations, then the EDO-paths are derived by concatenating the associated ESSs and DO-paths; (3) the executable back paths that connect EDO-paths' tail states to the initial state are generated, and (4) all of the executable test sequences (ETSs) are derived by concatenating the associated EDO-paths and their back paths, respectively.

References

[Chanson 93] S. T. Chanson and J. Zhu, "A Unified Approaches to Protocol Test Se-

- quence Generation", *Proc. of IEEE INFOCOM*, pp. 106-114, 1993.
- [Koh 94] L. S. Koh and M. T. Liu, "Test Path Selection Based on Effective Domains," *Proc. of International Conference on Network Protocols*, pp. 64-71, 1994.
- [Miller 92] R. E. Miller and S. Paul, "Generating Conformance Test Sequences for Combined Control and Data Flow of Communication Protocols," *Proc. of International Symposium on Protocol Specification, Testing and Verification, XII*, pp. 13-27, 1992.
- [Sarikaya 87] B. Sarikaya, G. V. Bochmann, and E. Cerny, "A Test Design Methodology for Protocol Testing," *IEEE Transactions on Software Engineering*, VOL. 13, NO. 5, pp. 518-531, 1987.
- [Ural 91] H. Ural and B. Yang, "A Test Sequence Selection Method for Protocol Testing," *IEEE Transactions on Communication*, VOL. 39, NO. 4, pp. 514-523, 1991.

Biography

Chung-Ming Huang received the B.S. degree in electrical engineering from National Taiwan University in 1984/6, and the M.S. and Ph.D. degrees in computer and information science from The Ohio state University, in 1987/12 and 1991/6, respectively. He is currently an associate professor in Institution of Information Engineering (IIE) National Cheng Kung University (NCKU), Taiwan, R.O.C. He is also the Chairman of ISO Special Interested Group (SIG) of Open Document Architecture (ODA) in Taiwan, R.O.C. He was the General Secretary of the Chinese Image Processing and Pattern Recognition Society from 1993/1 to 1994/12. His research interests include protocol testing and verification, multimedia networking protocols, and multimedia document systems.

Yuan-Chuen Lin received the B.S. degree in applied mathematics from Chung Yuan University in 1993/6, and had his master degree in IIE NCKU on 1995/6. His research interests include protocol testing and multimedia networking protocols.

Ming-Yuhe Jang received the B.S. degree in information computer engineering from Chung Yuan University in 1990/6, had qualifications to study directly Ph.D. degree in IIE NCKU in 1994/9, and is currently a Ph.D. student in IIE NCKU. His research interests include protocol testing, multimedia networking protocols, and ATM networking.