

Executable Test Sequence for the Protocol Data Portion Based on Two Criteria*

WEN-HUEI CHEN

Department of Electronic Engineering
Fu Jen Catholic University
Hsinchuang, 242 Taiwan
E-mail: paultaipei@yahoo.com.tw

A new test sequence generation method is proposed for testing the conformance of a protocol implementation to its data portion modeled by an *Extended Finite State Machine (EFSM)*, which is represented by a *Data Flow Digraph*. *All-Use* and *IO-df-chain* are two important criteria for selecting paths from the Data Flow Digraph to generate a test sequence which traces the data flow property, but it is a tedious process to select a path which satisfies the criteria while guaranteeing that the generated test sequence is *executable* (i.e., one that has feasible parameter values). In this paper, we propose a four-step method for generating the executable test sequence. The first step is a manual process, while the other three steps involves automatic as well as optimizing algorithms.

The first step involves the manual generation of a *Behavior Machine Digraph* from the Data Flow Digraph by embedding certain (but not all) parameter values so that every path of the new digraph is executable. Unlike the FSM for circuits, the size of the Data Flow Digraphs for real protocols can be managed test experts. In the second step, executable test paths which trace every association defined by each criterion are generated from the Behavior Machine Digraph. In the third step, the Behavior Machine Digraph is embedded with these test paths so as to construct the *SelectUse* and *SelectIO* Digraphs. Finally, the *Selecting Chinese Postman Tours* of the two digraphs are used to generate the executable test sequences that satisfy the All-Use and IO-df-chain criteria.

Keywords: conformance testing, executable test sequence, data flow, protocol, Chinese Postman tour

1. INTRODUCTION

A distributed system is composed of many *parties* (i.e., computers, instruments, etc.) remotely connected by communication *links* (i.e., cables, fibers, etc.) through which messages are transmitted. A *protocol* is the representation of as well as the orderly exchange of these messages that must be agreed on by any party before using it, and a set of protocols is usually layered to establish complex communicating behavior. In each party, a protocol is implemented in either software or hardware that has an upper (or lower) interface with the upper-layer (lower-layer) protocol(s) [9]. The objective of *protocol con-*

Received November 7, 2003; revised April 14 & May 25, 2004; accepted July 8, 2004.

Communicated by Chu-Sing Yang.

* This work was supported by the National Science Council of Taiwan under Grant NSC892213E030025 and by Fu Jen Catholic University under a grant from the SVD section. Part of this paper appeared as "Executable test sequence for the protocol data flow property" in IFIP 21st International Conference on Formal Techniques for Networked and Distributed Systems (FORTE), 2001.

formance testing is to see if a protocol implementation conforms to the protocol specification defined as a standard. In a testing center, the protocol implementation is tested as a black box. Inputs are sent from an external tester to the implementation through the interfaces, and the outputs are checked to see if they are as expected. The sequence of input/output pairs is the *test sequence*, and the number of inputs is the test sequence *length* [5]. In general, such a test sequence is generated from the protocol specification.

The protocol specification contains a control portion and a data portion [2]. The control portion determines how messages are sent and received. It can be considered a *deterministic*¹ *Finite State Machine* (FSM), which contains *states* and *transitions* [5]. Initially, the FSM is in a specific state, called the *initial state*. An input message (i.e., input or stimulus) will cause the FSM to generate output message(s) (i.e., outputs or responses) and to change from the current state to a new state; this process is called a *transition*. The data portion specifies other functions (e.g., quality of service) that involve the parameter values associated with the messages. Informally, the data portion is described in words which are then formulated as a set of rules among parameter values [6]. Formally, the data portion is specified by an *Extended Finite State Machine* (EFSM) [10], which is an extension from the FSM achieved by introducing *variables* and *parameters*. Initially, the EFSM is in the initial state, and all the variables are set to initial values. The EFSM can receive an input that has parameter values, which combine with certain variable values to define a logic function (i.e., *predicate*). That input will cause the EFSM to change from the current state to a new state that depends on the truth value of the predicate; then the EFSM will update the variable values according to a computation function (i.e., *action*) while generating output(s) that have certain parameter values.

In a data portion described by a set of rules, the generated test sequence is *executable* if it has parameter values that do not violate any rule [1, 6]. In [6], Kwast converted the FSM and rules into a *Behavior Machine Digraph*, in which paths can be used to generate the executable test sequence which verifies the rules. In [1], Chen converted the Behavior Machine Digraph into a *Selecting Digraph* and proposed the *Selecting Chinese Postman Algorithm* to find a specific tour of the digraph which generates a minimum-length executable test sequence that verifies each rule at least once. In [2], Chen generalized the Selecting Digraph to obtain a *SelectO Digraph* which is used to generate an executable test sequence that verifies both the control and data portions. However, the data portions of many protocols which are described in words cannot be formulated as rules. This paper is concerned with test sequence generation from the data portion specified by an EFSM.

The EFSM of a data portion can be represented by a Data Flow Digraph, where the inputs, outputs, predicates, and actions of the EFSM are represented by a set of nodes [11]. A test sequence generated from a path of such a digraph is *executable* (or *feasible*) if it has certain input parameter values which cause each predicate along the path to remain true. Because it is infeasible to traverse all the possible paths of the Data Flow Digraph, a test path is usually selected according to a criterion which involves a data flow property of the EFSM. In [12], a criterion was defined for observing the data flow abnormality of an EFSM due to a fault model. However, it is difficult for a test designer to construct a fault model that covers all the possible faulty implementations of an EFSM.

¹ Deterministic means that for each input there is at most one transition defined at each state.

In [10, 11], the *All-Use* and *IO-df-chain* criteria were defined for observing the data flow of variable values from where they are defined (or input) to where they are used (output). They have become important criteria in EFSM testing.

A test path of a Data Flow Digraph is said to satisfy the All-Use (or IO-df-chain) criterion if it can trace² all the variable value associations defined by that criterion. In order to trace an association, the test path starts from the first node (i.e., the initial state), proceeds to a specific node where a property of that association is exhibited, and then returns to the first node to produce in a *complete path* [11]. At the same time, we must ensure that the complete path allows the feasible assignment of input parameter values which cause all the predicates along the path to remain true. New complete paths must be obtained until that all the associations defined by the criterion have been traced. Unfortunately, no algorithms were proposed in [11] for efficiently constructing or concatenating these complete paths. The manual process is tedious and introduces a lot of overhead sequences in taking the EFSM to/from the initial state. In this paper, we will propose a four-step method for generating executable test sequences that satisfy the All-Use and IO-df-chain criteria. The first step is a manual process, while the other three steps involve automatic as well as optimizing algorithms.

We convert the Data Flow Digraph (the EFSM) into a *Behavior Machine Digraph*, the paths of which can be used to generate executable test sequences. Unlike the *Global FSM* which represents the complete behavior of the EFSM by enumerating all possible parameter and variable values as the inputs/outputs and states, the Behavior Machine only represents a partial behavior which will be used for testing. That is, only certain parameter (or variable) values are embedded into the inputs/outputs (states) for constructing the Behavior Machine where all predicates hold true [4]. Multiple paths of the Behavior Machine, Digraph trace the same variable value association defined by the criterion, but only one needs to be included in the final test sequence. Thus, we use the Selecting Chinese Postman Algorithm to optimally select either one to construct a minimum-length executable test sequence which traces each association at least once.

In section 2, the two criteria are reviewed. In section 3, the method based on the first criterion is proposed. In section 4, the method based on the second criterion is proposed. In section 5, our conclusions are presented.

2. THE TWO DATA FLOW TESTING CRITERIA

Consider the Simple Connection Protocol (SCN). The control portion specifies how to establish/release a connection. To establish a connection, an input "CONreq" (i.e., connection request) from the upper interface causes the SCN to output "connect" to the lower interface. "Accept" or "refuse" inputs from the lower interface cause the SCN to output "CONcnf(+)" (i.e., positive connection confirm) or "CONcnf(-)" (negative connection confirm) to the upper interface. After the connection is established, an input "Data" from the upper interface causes the SCN to output "data" to the lower interface. To release the connection, an input "Reset" from the upper interface causes the SCN to output "abort" to the lower interface. Notice that the uppercase (or lowercase) first letter indicates that the message is related to the upper (lower) interface.

² Such a trace is different from the trace used in verification to prove the property.

The data portion specifies other functions of the SCN by means of two types of variables. Variables of the first type are called the *memory variables*, which store temporary values. In the SCN, the memory variable TryCount stores the number of unsuccessful connection attempts and has values ranging from 0 to 2. The memory variables ReqQos and FinQos store the levels of requested and final quality of service, and have values from 0 to 3. Variables of the second type are called *parameter variables*, which store the parameter values of inputs. Let $X(y)$ denote an input X which has a parameter y ; then, the parameter variable denoted by $X.y$ will store the value of parameter y . This definition can be extended to multiple parameters. In the SCN, the parameter variables CONreq.qos, accept.qos and data.qos store the values of the parameter “qos” of inputs CONreq(qos), accept(qos), and data(qos), respectively. They have values ranging from 0 to 3.

The EFSM can be represented by the Data Flow Digraph shown in Fig. 1, where the states, inputs, outputs, and actions are represented by the *state nodes*, *input nodes*, *output nodes*, and *action nodes*, which enclose symbols that start with “s,” “i,” “o,” and “a,” respectively. Each predicate of the EFSM is represented by two *predicate-outcome nodes* (i.e., *predicate nodes*), which specify the two possible outcomes³. The predicate nodes enclose symbols that start with “p.” The execution sequence of nodes is indicated by edge directions. For example, consider Fig. 1. In the input node “i1”, the value of variable “CONreq.qos” will be input according to the statement “Input CONreq(qos).” Then, the value of “CONreq.qos” is decided to see whether it is larger than 1 or not. The EFSM may either arrive at either predicate node p1 or predicate node p2, which represent the two possible outcomes “CONreq.qos > 1” and “CONreq.qos ≤ 1.”

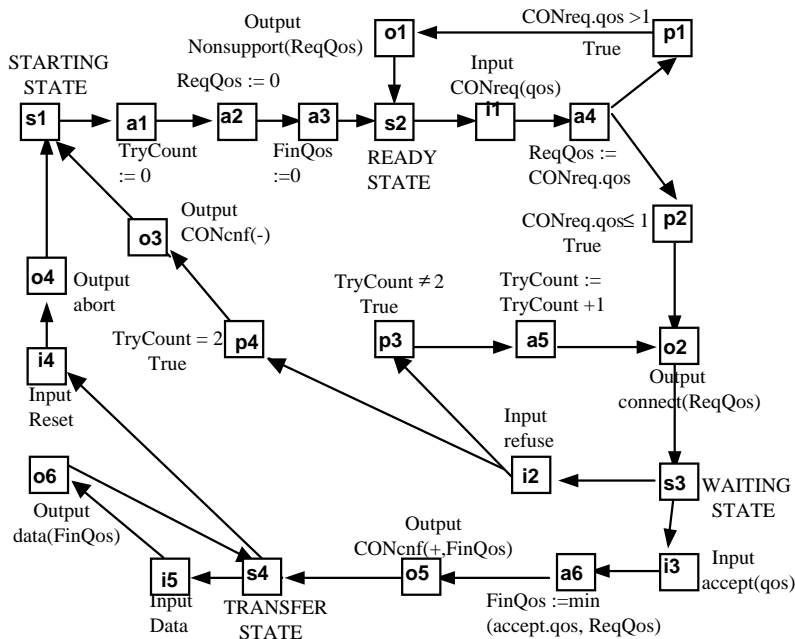


Fig. 1. The data flow digraph of the SCN protocol.

³ In [10, 11], a predicate was represented by a predicate node, and the results of the predicates were represented by edges leaving that node.

In Fig. 1, a *path* is a sequence of nodes that are connected by edges. A *tour* is a special path which starts and ends at the same node. A path can be used to generate a test sequence by considering only the inputs and outputs in the path. For example, the path $[s1, a1, a2, a3, s2, i1, a4, p1, o1, s2]$ is used to generate the test sequence [Input CONreq(qos), Output Nonsupport(qos)] (or simply [CONreq(qos)/Nonsupport(qos)]) by considering only input “ $i1$ ” and output “ $o1$.” In reality, the parameter “qos” must be given a value which makes predicate $p1$ hold true. For example, [CONreq(0)/Nonsupport(0)] is not executable because predicate $p1$ will not hold true, but [CONreq(2)/Nonsupport(2)] is executable. An *executable path* is a path that allows parameter values to be assigned so as to cause each predicate in the path to be true in order to generate the *executable test sequence*. Not all paths are executable. For example, the path $[s1, a1, a2, a3, s2, i1, a4, p2, o2, s3, i2, p4, o3, s1]$ is not executable. This is because the value of variable TryCount becomes “1” in node $a1$. In order to pass through predicate node $p4$, which requires that the value of the variable TryCount be equal to 2, the variable should increase in value in the intermediate nodes. However, the only node that can increment the value of TryCount is node $a5$, and that node is not in the path.

All-Use and IO-df-chain are two important criteria for selecting a test path from the Data Flow Digraph [11]. The first criterion claims that we should trace each variable from where it is *defined* (i.e., where its value is first assigned) to where that value is *used*. We will first describe where and when a variable is defined. At an input node, a variable is defined by an input statement. For example, at input node $i1$ in Fig. 1, variable CONreq.qos is defined by the input statement “Input CONreq(qos),” which gives variable CONreq.qos a value. At an action node, a variable is defined by a computation statement, which gives the variable a value. For example, at action node $a1$, the variable TryCount is defined by the statement “TryCount := 0,” where the variable TryCount is assigned the value of 0.

We will next explain where and when a variable is used. At a predicate node, a variable is used in the predicate statement where the value of the variable will determine the result of the predicate. For example, in Fig 1, variable CONreq.qos is used at predicate node $p1$ because the value of CONreq.qos will determine whether the predicate “CONreq.qos > 1” is true or not. At an action node, a variable is used in the computation statement where the variable value will determine the value of another variable. For example, at action node $a6$, the variable ReqQos is used in the statement “FinQos := min(accept.qos, ReqQos),” where the value of ReqQos determines the value of variable FinQos. At an output node, a variable is used in the output statement where the value of the variable is output. For example, at node $o2$, the variable ReqQos is used in the output statement “Output connect(ReqQos).” Table 1 lists the variables defined and used in Fig. 1.

Table 1. Variables defined and used at the nodes of the data flow digraph shown in Fig. 1.

Variable	Node where it is defined	Node where it is used
Conreq.qos	$i1$	$a4$ $p1$ $p2$
accept.qos	$i3$	$a6$
TryCount	$a1$ $a5$	$a5$ $p3$ $p4$
ReqQos	$a2$ $a4$	$a6$ $o2$ $o1$
FinQos	$a3$ $a6$	$o5$ $o6$

As a variable X is traced from node J (where X is defined) to node K (where X is used), the All-use criterion claims that the variable cannot not be redefined during the tracing process. Thus, to trace such an association, we must construct the *define-clear-use path*⁴ which connects node J to node K in such a way that the first node is the only node of the path where X is defined. If such a define-clear-path exists, we say that variable X and nodes J and K form an association called a *define-use pair* (or *du-pair*) $du(J, K, X)$. For example, consider Fig. 1 and Table 1. The variable ReqQos and nodes $a4$ and $o2$ form a du-pair $du(a4, o2, \text{ReqQos})$ because the variable can be traced by the define-clear-use path $[a4, p2, o2]$, where the variable ReqQos is defined exclusively at node $a4$ and used at node $o2$. It is possible that many define-clear-use paths can trace the same du-pair. But such a path may not exist at all so that not all the associations of variables and nodes form du-pairs. For example, the variable ReqQos and nodes $a2$ and $o2$ do not form a du-pair because the variable ReqQos is redefined in the intermediate node $a4$. A path (or test sequence) of the Data Flow Digraph is said to satisfy the All-Use criterion if all possible du-pairs can be traced.

The second criterion (i.e., the IO-df-chain criterion) claims that we should trace the data flow from a *first-defined node* J (where a variable X is first defined) to an output node K (where a variable Y is output), where X and Y can be either the same or different variables. Notice that J is a first-defined node if either; i) it is an input node where variable X is input a value or ii) it is an action node where variable X is assigned a constant value. For example, variable CONreq.qos is first defined at node $i1$ by the input statement “Input CONreq(qos),” and variable ReqQos is first defined at node $a2$ by the statement “ReqQos := 0,” which gives the variable a constant value “0.” Certainly, the value of variable X may not directly affect the value of variable Y , but it may affect it indirectly. That is, X may affect another variable that in turn affects variable Y , and so on. For example, consider the path $[i1, a4, p2, o2]$ in Fig. 1. At node $i1$, the EFSM receives an input CONreq which has a parameter qos, the value of which is stored by the parameter variable CONreq.qos. At node $a4$, the value of CONreq.qos affects the value of variable ReqQos. At node $o2$, the value of ReqQos affects the parameter value of output connect. As a result, we can control the parameter value of the input CONreq to observe the parameter value of the output connect. Obviously, a sequence of define-clear-use paths must be connected to trace this data flow.

By definition, a sequence of define-clear-use paths $p_1, p_2, \dots, p_{r-1}, p_r, \dots, p_s$ (for variables $X_1, X_2, \dots, X_{r-1}, X_r, \dots, X_s$) which starts from a node J (where the value of the variable X_1 is first defined) and ends at an output node K (where the value of the variable X_s is output) forms an input-output chain $IO(J, X_1, K, X_s)$. The chain $[p_1, p_2, \dots, p_{r-1}, p_r, \dots, p_s]$ can be used to trace the data flow from an node J where the variable X_1 is first defined to an output node K where the value of the variable X_s is output, because the value of variable X_r is determined by the value of variable X_{r-1} through a computation statement in the node where p_{r-1} and p_r intersect; i.e., the variable X_{r-1} affects X_r at the intersecting node (with the restriction that an action can have only one statement so that X_{r-1} cannot affect other variables). For example, $[i1, a4]$ is a define-clear-use path for the variable CONreq.qos, which starts from a node where CONreq.qos is first defined, and $[a4, p2, o2]$ is a define-clear-use path for the variable ReqQos, which ends at an output

⁴ The definition is an extension of the define-clear path of [10, 11].

node. The two paths are connected to form an io-chain $io(i1, \text{CONreq.qos}; o2, \text{ReqQos})$, which is $[i1, a4, p2, o2]$. A path (or test sequence) of the Data Flow Digraph is said to satisfy the IO-df-chain criterion if all possible io-chains are traced.

3. THE METHOD FOR THE FIRST CRITERION

In section 2, we described the All-Use and IO-df-chain criteria, which are guidelines for selecting the test paths from Data Flow Digraph. In this section, we will propose a method for generating the executable test sequence that satisfies the first criterion. Our method involves four steps. The first step is a manual process accomplished by a test expert who understands the protocol. The other three steps are conducted automatically.

In the first step, the test expert construct a set of du-pairs and produces a Behavior Machine Digraph (Fig. 2) from the Behavior Machine Digraph (Fig. 1), where every executable paths of Fig. 2 is executable. Unlike the FSMs for circuits, the size of the EFSM of a real protocol is not too large; thus, the test expert can construct these du-pairs (as described in [10, 11]) and the Behavior Machine Digraph by observing the functions of the protocol that he understands. When the Behavior Machine Digraph is constructed, a node J (see Fig. 1) is converted into nodes J_1, J_2, J_3, \dots (see Fig. 2)⁵ by embedding parameter values. If all parameter values are embedded, the Behavior Machine Digraph will be very large. Hence, the test expert will embed parameter values that cause the predicate node to be true. For example, consider node $i1$ (i.e., $\text{CONreq}(qos)$) in Fig. 1. The parameter qos has values ranging from 0 to 3, so we can convert node $\text{CONreq}(qos)$ into four nodes, $\text{CONreq}(0)$, $\text{CONreq}(1)$, $\text{CONreq}(2)$, and $\text{CONreq}(3)$. However, because either $\text{CONreq}(0)$ or $\text{CONreq}(1)$ can make predicate $p2$ true, we only create node $\text{CONreq}(1)$ (i.e., node $i1_2$), because the same input/output pair needs to be tested with one parameter. Similarly, either $\text{CONreq}(2)$ or $\text{CONreq}(3)$ can make predicate $p1$ true, so we only create node $\text{CONreq}(2)$ (i.e., node $i1_1$). Because every predicate in the digraph is true, any path of the digraph can be used to generate an executable test sequence.

In Fig. 2, the Behavior Machine is represented by fine lines and text, but it includes some bold characters that are used in the next step to trace the data flow. In general, a bold character " X_{def} " (or " X_{use} ") is added to nodes J_1, J_2, J_3, \dots in Fig. 2 if variable X is defined (used) at node J in Fig. 1. For example, a bold character C_{def} (an abbreviation of $\text{CONreq.qos}_{\text{def}}$) is added to nodes $i1_1$ and $i1_2$ in Fig. 2 because the variable CONreq.qos is defined at node $i1$ in Fig. 1 (see Table 1).

The second step involves finding executable test paths in Fig. 2 that can trace the du-pairs in Table 1, as indicated in Table 2. Consider the du-pair $du(J, K, X)$, where variable X is defined at node J and used at node K . As described in section 2, this du-pair is traced by a define-clear-use path in Fig. 1, which starts from node J and ends at node K and does not redefine the variable in the intermediate nodes. In Fig. 2, such an executable define-clear-use path corresponds to specific paths which start from nodes $J_1, J_2, \dots, J_r, \dots, J_s$ and end at nodes $K_1, K_2, \dots, K_p, \dots, K_q$, where the symbol X_{def} can only be seen at the starting nodes. These specific paths in Fig. 2 can be constructed from the shortest

⁵ Without loss of generality, the notations J_1, J_2, J_3, \dots of the Behavior Machine Digraph represent the node10s converted from node J of the Data Flow Digraph in this paper.

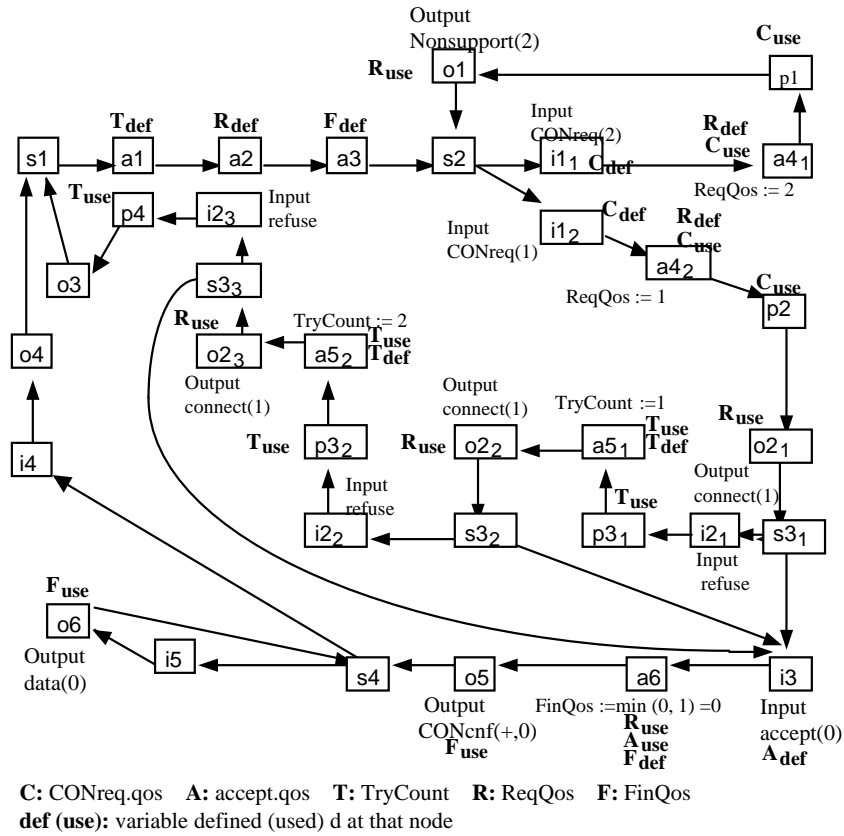


Fig. 2. The behavior machine digraph of the data flow digraph shown in Fig. 1 (only new statements other than those in Fig. 1 are indicated).

Table 2. Executable test paths of Fig. 2 for tracing the du-pairs of Table 1.

Label	du-pairs in Table 1	Executable define-clear-use paths in Fig. 2
1	$du(i1, a4, \text{CONreq.qos})$	$[i1_1, a4_1]$ or $[i1_2, a4_2]$
2	$du(i1, p1, \text{CONreq.qos})$	$[i1_1, a4_1, p1]$
3	$du(i1, p2, \text{CONreq.qos})$	$[i1_2, a4_2, p2]$
4	$du(i3, a6, \text{accept.qos})$	$[i3, a6]$
5	$du(a1, a5, \text{TryCount})$	$[a1, a2, a3, s2, i1_2, a4_2, p2, o2_1, s3_1, i2_1, p3_1, a5_1]$
6	$du(a1, p3, \text{TryCount})$	$[a1, a2, a3, s2, i1_2, a4_2, p2, o2_1, s3_1, i2_1, p3_1]$
7	$du(a5, p3, \text{TryCount})$	$[a5_1, o2_2, s3_2, i2_2, p3_2]$
8	$du(a5, p4, \text{TryCount})$	$[a5_2, o2_3, s3_3, i2_3, p4]$
9	$du(a4, o2, \text{ReqQos})$	$[a4_2, p2, o2_1]$ or $[a4_2, p2, o2_1, s3_1, i2_1, p3_1, a5_1, o2_2, s3_2, i2_2, p3_2, a5_2, o2_3]$
10	$du(a4, a6, \text{ReqQos})$	$[a4_2, p2, o2_1, s3_1, i3, a6]$
11	$du(a4, o1, \text{ReqQos})$	$[a4_1, p1, o1]$
12	$du(a6, o5, \text{ReqQos})$	$[a6, o5]$
13	$du(a6, o6, \text{ReqQos})$	$[a6, o5, s4, i5, o6]$

paths in Fig. 2, where nodes which have the symbol X_{def} (except the starting nodes) and their adjacent edges are (temporarily) removed. For example, to trace the du-pair $du(i1, a4, CONreq.qos)$ in Table 1, where CONreq is defined at node $i1$ and used at node $a4$, we find the specific shortest paths which start at nodes $i1_1$ and $i1_2$ and end at nodes $a4_1$ and $a4_2$ in Fig. 2. These shortest paths are used to produce the executable define-clear-use paths $[i1_1, a4_1]$ and $[i1_2, a4_2]$ that can trace the du-pair $du(i1, a4, CONreq.qos)$. The complete executable define-clear-use paths in Fig. 2 for tracing the du-pairs in Table 1 are shown in Table 2.

The third step involves constructing a SelectUse Digraph in Fig. 3 by embedding the executable test paths in Table 2 into the Behavior Machine Digraph in Fig. 2.

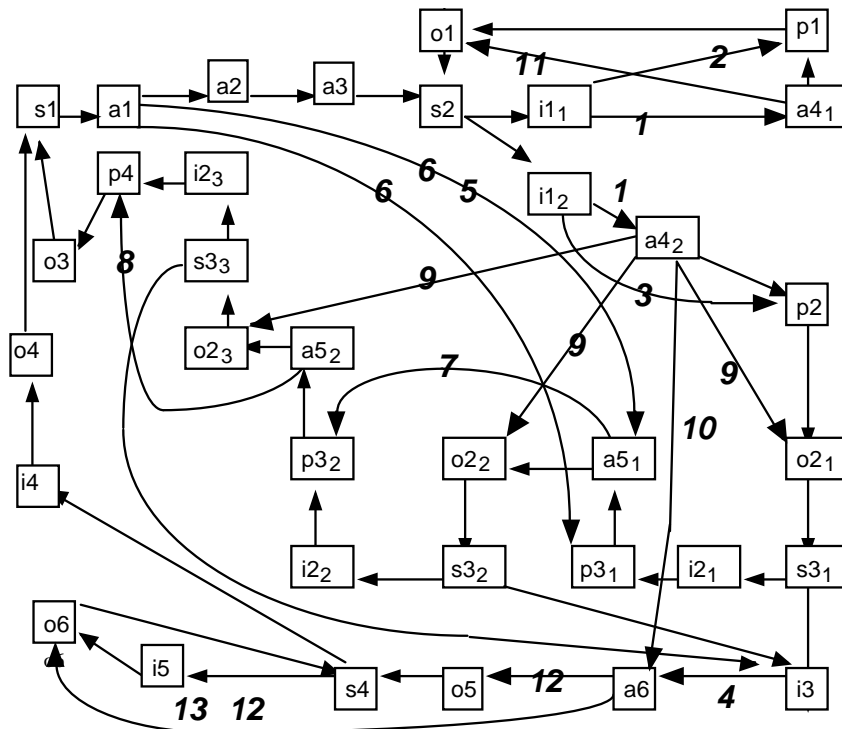


Fig. 3. The SelectUse digraph constructed from the behavior machine digraph in Fig. 2 by embedding the define-clear-use paths in Table 2 as bold edges.

Generally, an executable define-clear-use path from node J to node K is embedded as a bold edge from node J to node K , and a bold label is added the path to indicate the du-pair traced by the path. We assume that each input will take the same testing time; thus, we assign a cost to an edge according to the number of inputs represented by that edge. The case where an input takes non-uniform testing time will be discussed in section 5. Thus, a fine edge is assigned a cost of 1 if it leaves an input node because it represents only one input, and a bold edge is assigned a cost which reflects the number of inputs

contained in the define-clear-use path represented by the bold edge. For example, consider the executable test path $[a1, a2, a3, s2, i1_2, a4_2, p2, o2_1, s3_1, i2_1, p3_1, a5_1]$ in Table 2 that traces the du-pair $du(a1, a5, \text{TryCount})$, which has the label 5. Because the path starts at node $a1$ and ends at node $a5_1$, it is embedded as a bold edge from node $a1$ to $a5_1$. The bold edge is labeled 5 and assigned a cost of 2. As stated in section 1, the number of inputs is the test sequence *length*, a minimum-cost path of this digraph, where edges are assigned costs which correspond to a minimum-length executable test sequence.

An executable define-clear-use path that can trace multiple du-pairs is embedded as an edge that has multiple labels. For example, the define-clear-use path described above can trace two du-pairs in Table 2, and we add labels 5 and 6 on it. For many bold edges that share the same label, only one edge needs to be included in the final test path because these bold edges represent the executable test paths that trace the same du-pair. As a result, the Selecting Chinese Postman Tour of the SelectUse Digraph (which is a minimum-cost tour, where each type of label appears at least once) can be used to generate a minimum-length executable test sequence that traces all the du-pairs.

The fourth step involves using the Selecting Chinese Postman Algorithm proposed in [1] to find the Selecting Chinese Postman Tour of the SelectUse Digraph in order to generate the executable test sequence. The algorithm contains two phases. The first phase replicates (or deletes) each edge of the SelectUse Digraph, resulting in a *minimum-cost Selecting Symmetric Augmentation* (MCSSA) which satisfies the following properties; i) each node has the same number of entering and leaving edges, and ii) each type of label appears in the digraph at least once. Labels belong to the same type if they are represented by the same character (see Table 2). The MCSSA can be obtained by solving a system of integer programming equations formulated from the SelectUse Digraph. The integer programming problem is solved by a branch and bound algorithm which iterates in order to improve an initial solution [8]. When the problem size is not very large, the branch and bound algorithm can find the optimal solution. Otherwise, the branch and bound algorithm will at least obtain a significantly improved solution if we can make the algorithm run a long time. In the second phase, we check whether the MCSSA is an Euler Digraph or a collection of disjoint Euler Digraphs [7]. In the former case, a Euler Tour of the Euler Digraph is a Selecting Chinese Postman Tour of the SelectUse Digraph. In the latter case, a tour will be used to connect these disconnected Euler Digraphs to form a Euler Digraph, so that the Euler Tour algorithm can be applied.

Consider the example shown in Fig. 3; the Lindo package [8] has solved the integer programming equations in less than one second and results in a Euler Digraph (i.e., the first case), so that the optimal tour is found. The Selecting Chinese Postman Tour in Fig. 3, $[s1, a1, a2, a3, s2, i1_1, \mathbf{1}, a4_1, \mathbf{11}, o1, s2, i1_1, \mathbf{2}, p1, o1, s2, i1_2, \mathbf{1}, a4_2, \mathbf{10}, a6, (\mathbf{13}, \mathbf{12}), o6, s4, i4, o4, s1, a1, a2, a3, s2, i1_2, \mathbf{3}, p2, o2_1, s3_1, i3, \mathbf{4}, a6, \mathbf{12}, o5, s4, i4, o4, s1, a1, a2, a3, s2, i1_2, \mathbf{1}, a4_2, \mathbf{9}, o2_2, s3_2, i2_2, p3_2, a5_2, o2_3, s3_3, i2_3, p4, o3, s1, a1, (\mathbf{6}, \mathbf{5}), a5_1, \mathbf{7}, p3_2, a5_2, \mathbf{8}, p4, o3, s1, a1, a2, a3, s2, i1_2, \mathbf{3}, p2, o2_1, s3_1, i3, \mathbf{4}, a6, \mathbf{12}, o5, s4, i4, o4, s1]$ is used to generate the minimum-length executable test sequence $[i1_1, o1, i1_1, o1, i1_2, o2_1, i3, o5, i5, o6, i4, o4, i1_2, o2_1, i3, o5, i4, o4, i1_2, o2_1, i2_1, o2_2, i2_2, o2_2, i2_3, o3, i1_2, o2_1, i2_1, o2_2, i2_2, o2_3, i2_3, o3, i1_2, o2_1, i3, o5, i4, o5]$, which traces all the du-pairs in Table 1.

Formally, the algorithm for steps 2 to 4 is described as follows. Inputs to Algorithm 1 are produced in step 1 performed by the test expert.

Algorithm 1

Input: A Behavior Machine Digraph $G'(V', E')$ and a set of du-pairs.

Output: An executable test sequence which traces each du-pair at least once.

Step 2: Find executable test paths to trace du-pairs.

For each du-pair (J, K, X)

Remove nodes labeled with “ X_{def} ” and their adjacent edges from $G'(V', E')$ to produce in a temporary digraph $G^t(V^t, E^t)$;

Find shortest paths from J_p to k_q for all p, q in digraph $G^t(V^t, E^t)$ to produce test paths for tracing define-use pair (J, K, X) ;

Number these test paths in alphabetical order (a number is considered to be a label of a path);

Step 3: Construct the SelectUse Digraph

For each executable test path from node J_p to node k_q which has a label L

Add a bold edge $(J_p, K_q; L)$ to $G'(V', E')$ to produce in $G''(V'', E'')$;

Cost of edge $(J_p, K_q; L) = \text{length of the test path}$;

For each fine edge e of $G''(V'', E'')$

Cost of edge $e = 1$;

Step 4: Find the Selecting Chinese Postman Tour of $G''(V'', E'')$ (see reference [1])

4.1 Solve the Integer Programming Equations

Solve the following equations for each edge e_j of E'' where $j = 1, 2, \dots, r$:

Minimize $X(e_1) * \text{Cost}(e_1) + \dots + X(e_{r-1}) * \text{Cost}(e_{r-1}) + X(e_r) * \text{Cost}(e_r)$

subject to

$$\forall \text{ vertex } p, \sum_{e_j \text{ enters } p} X(e_j) = \sum_{e_k \text{ leaves } p} X(e_k)$$

$$\forall \text{ vertex } q, \sum_{\substack{e_j \text{ where label } q \\ \text{appears}}} X(e_j) \geq 1$$

\forall edge $e_j, X(e_j), X(e_j)$ is an integer.

4.2 Construct the minimum-cost Selecting Symmetric Augmentation of $G''(V'', E'')$

For each edge e_j of E''

If $X(e_j) = 0$

remove edge e_j ;

else

replicate edge e_j into $X(e_j)$ edges that have the same cost and same label;

endif

4.3 Generate the tour

Use the algorithm in [13] to check if the Augmentation is connected;

If the Augmentation is not connected

Find a tour C of the Augmentation which passes through every connected component;

Add C to $G''(V'', E'')$;

endif

Use the algorithm in [13] to generate the Euler tour of $G'(V', E')$;

Convert the tour into an executable test sequence

We will now analyze the complexity of this algorithm. The complexity of step 2 is $O(t * |V'| * (|E'| + |V'| \log |V'|))$, where t is the number of du-pairs determined in [10, 11], and $(|E'| + |V'| \log |V'|)$ is based on the shortest path algorithm [13]. Fortunately, the sizes of the Behavior Machine Digraphs of real protocols are not too large. The complexity of step 3 is $O(|E''|) = O(|V''|^2) = O(|V'|^2)$ (because $|V'| = |V''|$ as the SelectUse Digraph only adds edges to the Behavior Machine Digraph but does not increase the number of vertices). The complexity of step 3 is analyzed in [1].

We also present the following lemmas.

Lemma 1 Step 2 of Algorithm 1 can be used to generate the shortest executable test paths to trace the du-pairs.

Proof: We will consider a Behavior Machine Digraph G^* which is obtained from the Data Flow Digraph by embedding all the parameter values; the shortest paths of G^* can clearly be used to generate executable test paths which trace the du-pairs. A practical Behavior Machine Digraph G' differs from G because edge e^* is removed if there is already an edge e , where e and e' come from the same vertex, end at the same vertex, and cause the predicate to be true. Thus, step 2 will obtain the shortest paths of G' , which are also the shortest paths of G^* . Hence, step 2 can be used to generate the shortest executable test paths to trace the du-pairs.

Lemma 2 Steps 3 and 4 of Algorithm 1 can be used to generate the shortest executable test sequence, which includes each shortest executable test path that traces each du-pair at least once if the Augmentation of step 4.3 is connected.

Proof: If the Augmentation of step 4.3 is connected, step 4 will obtain the optimal Selecting Chinese Postman Tour (see reference [1] for the proof). Consider the Behavior Machine Digraph G^* which is obtained from the Data Flow Digraph by embedding all the parameter values. Similar to the proof of Lemma 1, the optimal Selecting Chinese Postman Tour of G' will correspond to an optimal Selecting Chinese Postman Tour of G^* , which can be used to generate the shortest executable test sequence that includes each shortest executable test path which traces each du-pair at least once.

Lemma 3 If the Augmentation of step 4.3 is not connected, the obtained executable test sequence will be longer than the optimal one by a bound of the cost of C , which is the tour added to make the Augmentation connected.

Proof: It directly follows from [1].

4. THE METHOD FOR THE SECOND CRITERION

In section 3, we proposed a method to produce an executable test sequence which satisfies the first criterion by finding the Selecting Chinese Postman Tour [1] of the SelectUse Digraph. In this section, we will extend the method to generate an executable test

sequence which satisfies the second criterion, i.e., the IO-df-chain criterion which requires the tracing of each io-chain at least once. The extended method includes three steps.

The first step involves finding all the io-chains in Fig. 1. In the SelectUse Digraph in Fig. 3, we consider nodes J_1, J_2, J_3, \dots , (converted from node J , where variable X is first defined) to output nodes K_1, K_2, K_3, \dots (converted from node K , where variable Y is output). As described in section 2, an io-chain is constructed from a sequence of define-clear-use paths so as to connect a first-defined node (i.e., a node where a variable is assigned a constant value) to an output node (but notice that a sequence of define-clear paths as defined in [11] does not necessarily form an input-output-chain). In Fig. 3, the bold edges that leave first-defined nodes J_1, J_2, J_3, \dots , which are the define-clear-use paths for variable X , are called $(X - J)$ edges, and those bold edges which enter output nodes K_1, K_2, K_3, \dots , which are the define-clear-use paths for variable Y , are called $(Y - K)$ edges. A path which involves only bold edges is a bold path. In the SelectUse Digraph, the shortest bold path from the $(X - J)$ edges to the $(Y - K)$ edges can be used to generate the sequence of define-clear-use paths which can trace io-chain $io(J, X, K, Y)$. The specific shortest paths of the SelectUse Digraph can be obtained from the shortest paths of the SelectUse Digraph, where fine edges are (temporarily) removed. For example, the bold edge (i.e., the define-clear-use paths $[i1_1, a4_1]$) and another bold edge (i.e., $[a4_1, II, o1]$) compose an io-chain $io(i1, \text{CONreq.qos}, o1, \text{ReqQos}) = [i1_1, I, a4_1, II, o1]$. The complete io-chains for Fig. 1 obtained from these specific shortest paths in Fig. 3 are shown in Table 3, where each io-chain is given a label.

Table 3. The io-chains for the data flow digraph in Fig. 1.

Label	Input-output-chains (Fig. 1)	Composed define-clear-use paths (Fig. 3)	Corresponding executable paths (Fig 2)
a	$io(i1, \text{CONreq.qos}, o1, \text{ReqQos})$	$[i1_1, I, a4_1, II, o1]$	$[i1_1, a4_1, p1, o1]$
b	$io(i1, \text{CONreq.qos}, o2, \text{ReqQos})$	$[i1_2, I, a4_2, 9, o2_1]$	$[i1_2, a4_2, p2, o2_1]$
		$[i1_2, I, a4_2, 9, o2_2]$	$[i1_2, a4_2, p2, o2_1, s3_1, i2_1, p3_1, a5_1, o2_2]$
		$[i1_2, I, a4_2, 9, o2_3]$	$[i1_2, a4_2, p2, o2_1, s3_1, i2_1, p3_1, a5_1, o2_2, s3_2, i2_2, p3_2, a5_2, o2_3]$
c	$io(i1, \text{CONreq.qos}, o5, \text{FinQos})$	$[i1_2, I, a4_2, 10, a6, 12, o5]$	$[i1_2, a4_2, p2, o2_1, s3_1, i3, a6, o5]$
d	$io(i1, \text{CONreq.qos}, o6, \text{FinQos})$	$[i1_2, I, a4_2, 10, a6, 13, o6]$	$[i1_2, a4_2, p2, o2_1, s3_1, i3, a6, o5, s4, i5, o6]$

The second step involves constructing the SelectIO Digraph in Fig. 4 by embedding the executable test paths in Table 3 into the Behavior Machine Digraph in Fig. 2. Generally, an executable test path listed in Table 3 which starts from input node J and ends at output node K is embedded as a bold edge from vertex J to vertex K . The label which represents the io-chain traced by the executable path is put on the edge (see Table 3 and

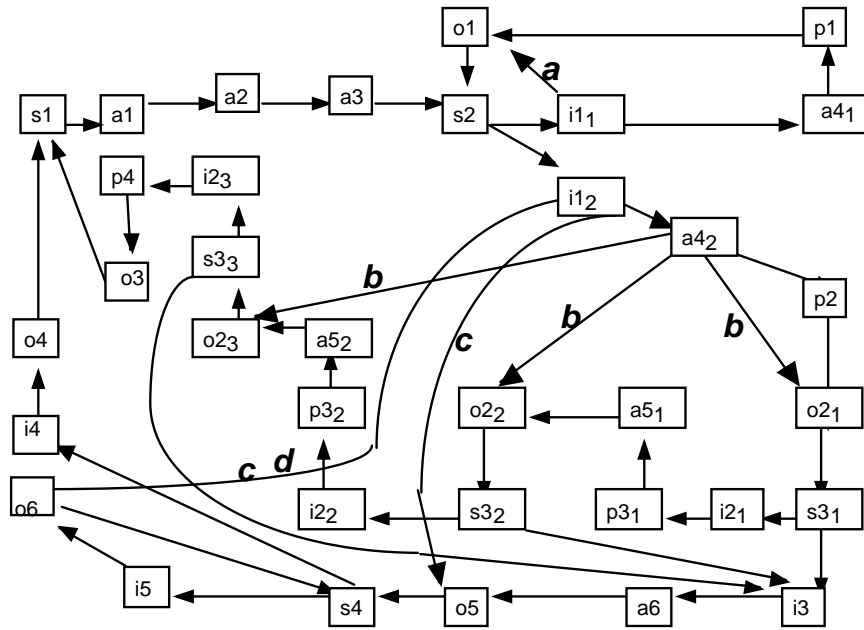


Fig. 4. The SelectIO digraph constructed from the behavior machine digraph in Fig. 2 by embedding the io-chains in Table 3 as bold edges.

Fig. 4). Costs are assigned to the edges of the SelectIO Digraph similar to the process described in the third step discussed in section 3 for assigning costs to the edges of the SelectUse Digraph.

The third step involves using a Selecting Chinese Postman Tour of the SelectIO Digraph to generate a minimum-cost executable test sequence that checks each io-chain. The tour is obtained using the algorithm described in the fourth step discussed in section 3. The Selecting Chinese Postman Tour in Fig. 4, $[s1, a1, a2, a3, s2, i11, a, o1, s2, i12, b, o21, s31, i21, p31, a51, o22, s32, i22, p32, a52, o23, s33, i23, p4, o3, s1, a1, a2, a3, s2, i12, (c, d), o6, s4, i4, s1]$, is used to generate the executable test sequence $[i11, o1, i12, o21, i21, o22, i22, o23, i23, o3, i12, o21, i3, o5, i5, o6, i4, o4]$, which traces all the io-chains in Table 3.

Formally, the above process is described as follows:

Algorithm 2

Input: The SelectUse Digraph $G''(V'', E'')$, the Behavior Machine Digraph $G'(V', E')$.

Output: An executable test sequence where traces each io-chain at least once.

Step 1: Find the shortest io-chains

Remove all fine edges from $G''(V'', E'')$ to produce in a temporary digraph to produce in a temporary digraph $G'(V', E')$ which contains only bold edges;

Find the shortest paths from the $(X - J)$ edges to the $(Y - K)$ edges to generate the executable test path which traces io-chain $io(J, X, K, Y)$;

Step 2: Construct the SelectIO Digraph

For each executable test path from node J_p to node k_q which has a label L
 Add a bold edge $(J_p, K_q; L)$ to $G''(V'', E'')$ to produce in $G'''(V''', E''')$;
 Cost of edge $(J_p, K_q; L) = \text{length of the test path}$;
 For each fine edge e of $G''(V'', E'')$
 Cost of edge $e = 1$;

Step 3: Use the algorithm in [1] (see also step 4 of Algorithm 1) to generate the Selecting Chinese Postman Tour of $G'''(V''', E''')$ and use it to generate the executable test sequence that traces each io-chain at least once.

We want to analyze the complexity of this algorithm. Step 1 needs $O(|V''|^3) = O(|V'|^3)$. Step 2 needs $O(|E''|) = O(|V''|^2)$. The complexity of step 3 was analyzed in [1].

Similar to the proofs of Lemmas 1, 2 and 3 given in section 3, we have the following lemmas.

Lemma 4 Step 1 of Algorithm 2 can be used to generate the shortest executable test paths to trace the io-chains.

Lemma 5 Steps 2 and 3 of Algorithm 2 can be used to generate the shortest executable test sequence which includes each shortest executable test path that traces each du-pair at least once if the Augmentation of step 3.3 is connected.

Lemma 6 If the Augmentation of step 3.3 is not connected, the obtained executable test sequence will be longer than the optimal one by a bound of the cost of C , which is the tour added to make the Augmentation connected.

5. CONCLUSIONS

In this paper, we have proposed a method to automatically generate an executable test sequence from a protocol specification to verify two data flow criteria, based on finding the Selecting Chinese Postman Tour of the SelectUse and SelectIO Digraphs constructed from the Data Flow Digraph of the protocol. Compared with our earlier method [1] based on the Behavior Machine Digraph, this new method is superior because the earlier one was based on a data portion of rules that can not specify most protocols, while the current one is based on a Data Flow Digraph which can specify most protocols. Compared with to the test sequence generation method [10, 11] based on the data portion represented by the Data Flow Digraph, this news method is superior because the former is completely manual and involves no optimization techniques, while only the first step of the new method is manual, and we have provided several optimization algorithms.

The first phase of the Selecting Chinese Postman Algorithm involves solving a system of integer programming equations so as to find an augmentation. From our experience with solving such equations [1, 2], a linear programming version of the formulation always yields integer results. We want to check whether these equations satisfy a specific property so that the linear programming approach can be applied. Our method for minimizing the test sequence length can be easily extended to one minimizing the test se-

quence cost by assigning costs to the edges of the digraph. In addition, the method can be combined with the *DuplexE Digraph* method to generate a synchronizable and executable test sequence [3].

Our method produces an executable test sequence, where the define-clear-use paths or input-output chains do not overlap. Our method is not completely automatic because the first step involves manual generation of the Behavior Machine Digraph by a test expert. In [4], an automatic algorithm was proposed for generating the minimal Behavior Machine Digraph from an EFSM that involves no parameters. In the future, we aim to extend that algorithm for the EFSM that involves parameters. We will also investigate techniques that can be used to overlap these paths (chains) to obtain an even shorter test sequence.

REFERENCES

1. W. H. Chen, "Test sequence generation from the protocol data portion based on the selecting Chinese postman algorithm," *Information Processing Letters*, Vol. 65, 1998, pp. 261-268.
2. W. H. Chen, "Executable test sequence for the protocol control and data portions," in *Proceedings of IEEE International Conference on Communications*, 2000, pp. 505-510.
3. W. H. Chen and H. Ural, "Synchronizable test sequence based on multiple UIO sequences," *IEEE/ACM Transactions on Networking*, Vol. 3, 1995, pp. 152-157.
4. K. T. Cheng and A. S. Krishnakumer, "Automatic functional test generation using the extended finite state machine model," in *Proceedings of IEEE Design Automation Conference*, 1993, pp. 86-91.
5. R. Dssouli, K. Saleh, E. Aboulhamid, A. En-Nouaary, and C. Bourhfir, "Test development for communication protocols: towards automation," *Computer Networks*, Vol. 31, 1999, pp. 1835-1872.
6. E. Kwast, "Towards automatic test generation for protocol data aspects," in *Proceedings of IFIP International Symposium on Protocol Specification, Testing, and Verification*, 1992, pp. 333-348.
7. J. A. Mchugh, *Algorithmic Graph Theory*, Prentice-Hall, Englewood Cliffs, New Jersey, 1990.
8. L. Schrage, *LINDO 5.0 Users Manual*, Scientific Press, 1991.
9. M. Schwartz, *Telecommunication Networks: Protocols, Modeling and Analysis*, Addison-Wesley Publishing Company, 1987.
10. H. Ural, "Test sequence selection based on static data flow analysis," *Computer Communications*, Vol. 10, 1987, pp. 234-242.
11. H. Ural, K. Saleh, and A. Williams, "Test generation based on control and data dependencies within system specification in SDL," *Computer Communications*, Vol. 23, 2000, pp. 609-627.
12. C. J. Wang and M. T. Liu, "Generating test cases for EFSM with given fault models," in *Proceedings of IEEE INFOCOM*, 1993, pp. 774-781.
13. J. A. Mchugh, *Algorithmic Graph Theory*, Prentice-Hall, Englewood Cliffs, New Jersey, 1990.



Wen-Huei Chen (陳文輝) received his B.S. degree in Electrical Engineering from National Sun Yat-Sen University, Kaohsiung, Taiwan in 1986, his M.S. degree in Electrical Engineering from National Taiwan University, Taipei, Taiwan in 1988 and his Ph.D. from National Tsing Hua University, Hsinchu, Taiwan in 1994. Since 1994, he had been an associate professor in the Department of Information Management, Ming Chuan University, Taipei, Taiwan. Since 2000, he has been an associate professor in the Department of Electronic Engineering, Fu Jen Catholic University, Taipei, Taiwan. His current research interests include network security and protocol engineering.