EXECUTABLE TEST SEQUENCE FOR THE PROTOCOL DATA FLOW PROPERTY[1]

Wen-Huei (Paul) Chen
Department of Electronic Engineering, Fu Jen Catholic University, Taipei, Taiwan, R.O.C.
paultaipei@yahoo.com.tw

Abstract 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* which is represented by a *Data Flow Digraph. All-Use* and *IO-df-chain* are two important criteria in selecting paths from the Data Flow Digraph in order 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.) The proposed four-step method automatizes the selecting process as well as optimizing the test sequence length. First, the Data Flow Digraph is embedded with certain (but not all) parameter values so as to construct a *Behavior Machine Digraph* where executable test sequences can be directly generated. Second, executable test paths which trace every association defined by each criterion are generated from the Behavior Machine Digraph. Third, 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 optimally executable test sequences that satisfy the All-Use and IO-df-chain criteria.

Keywords: Conformance testing, executable test sequence, data flow.

1. INTRODUCTION

A distributed system is composed of many *parties* (i.e., computers, instruments, etc.) remotely connected by communication *links* (i.e., cables,

[1] This work is supported by the National Science Council of Taiwan under Grant NSC892213E030025 and by Fu Jen University under a grant from the SVD section.

fibers, etc.) through which messages are transmitted, A *protocol* is the representation 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 a complex communicating behavior. In each party, a protocol is implemented in either software or hardware that has an upper (or lower) interface to the upper-layer (lower-layer) protocol(s) [9]. The objective of *protocol conformance 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 and a data portion [2]. The control portion determines how messages are sent and received. It can be considered a *deterministic*[2] *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 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 into a set of rules among parameter values [6]. Formally, the data portion is specified by an *Extended Finite State Machine (EFSM)* [2] which is extended from the FSM 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 updates 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 converts 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 converts the Behavior Machine Digraph into

[2] "deterministic" means that for each input there is at most one transition defined at each state.

a *Selecting Digraph* and proposes 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 generalizes the Selecting Digraph into 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 into 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 the digraph is *executable* (or *feasible*) if it has certain input parameter values which make each predicate along the path to remain true. Because it is infeasible to traverse all 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 is defined for observing the data flow abnormality of the EFSM due to a fault model. However, it is difficult for a test designer to construct a fault model that covers all possible faulty implementations of the EFSM. In [10][11], the *All-Use* and *IO-df-chain* criteria are 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 the Data Flow Digraph is said to satisfy the All-Use (or IO-df-chain) criterion if it can trace all the variable value associations defined by that criterion. In order to trace[3] an association, the test path starts from the first node (i.e., the initial state) and proceeds to a specific node where a property of that association is exhibited, then returns to the first node to result in a *complete path* [11]. At the same time, we must ensure that the complete path allows a feasible assignment of input parameter values which makes all predicates along the path remain true. New complete paths must be obtained until that all associations defined by the criterion are traced. Unfortunately, this process is tedious and introduces a lot of overhead sequences in taking the EFSM to/from the initial state. In this paper, we are going to propose an automatic method to generate the executable test sequences that satisfy the All-Use and IO-df-chain criteria as well as optimizing the test sequence length.

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

[3] such a trace is different from the "trace" used in verification for proving the property.

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 are removed [4]. For instance, the Simple Connection Protocol of 4 states (see next section) is converted to a Global FSM of 768 states but is converted to a Behavior Machine of 6 states. Multiple paths of the Behavior Machine Digraph trace the same variable value association defined by the criterion, but only one needs to be included into 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. Inputs "accept" or "refuse" from the lower interface causes the SCN to output "CONrsp(+)" (i.e., positive connection response) or "CONrsp(-)" (negative connection response) 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 cause 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.

The data portion specifies other functions of the SCN by two types of variables. Variables of the first type are called the *memory variable* that will store temporary values. In the SCN, the memory variable TryCount stores the number of unsuccessful connection attempts and has values from 0 to 2, and 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* that will 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 will

store the values of the parameter "qos" of inputs "CONreq(qos)", "accept(qos)" and "data(qos)" respectively. They have values from 0 to 3.

The EFSM can be represented by the Data Flow Digraph of Figure 1,

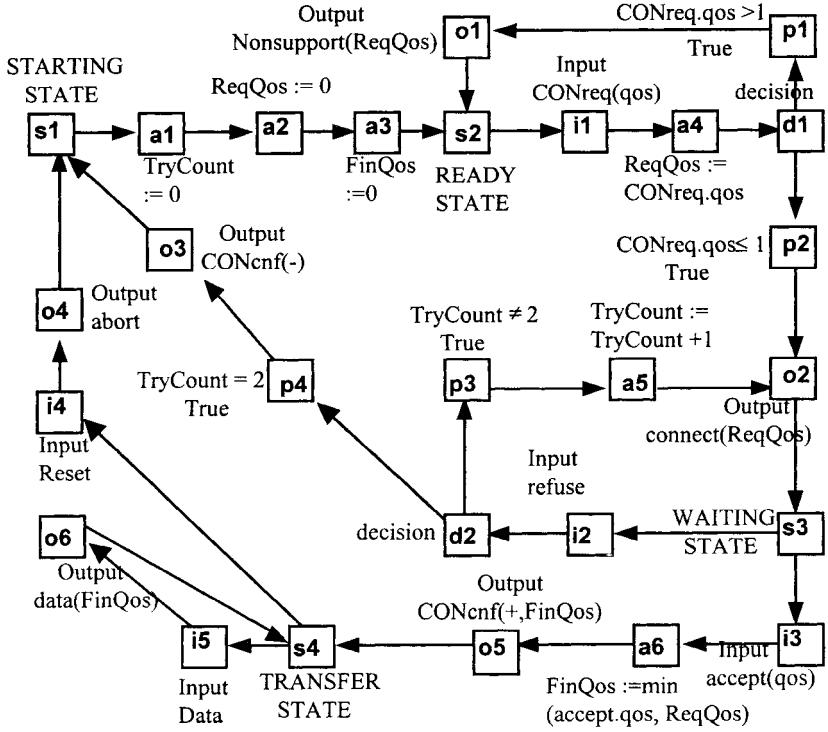


Figure 1. The Data Flow Digraph of the SCN protocol

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 one predicate-decision node (i.e., *decision node*) and two predicate-outcome nodes (i.e., *predicate nodes*) which specify the two possible outcomes [4]. The decision and predicate nodes enclose symbols that start with "d" and "p" respectively. The execution sequence of nodes is indicated by edge directions. For example, consider Figure 1. In the input node "i1", the value of variable CONreq.qos will be input according to the statement "Input CONreq(qos)." Then, in decision node "d1", the value of CONreq.qos is decided to see whether it is larger than 1 or not. Then, the EFSM may either arrive at either the predicate node p1 or predicate node p2

[4] In [10][11], a predicate is represented by a predicate node and the results of the predicates are represented by edges leaving that node.

which represents the two possible outcomes "CONreq.qos >1" and "CONreq.qos ≤ 1."

In Figure 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, d1, 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 which causes 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, d1, p2, o2, s3, i2, d2, p4, o3, s1] is not executable. It is because that the value of variable TryCount becomes "1" in node a1. In order to pass through predicate node p4 which requires that the value of variable TryCount be equal to 2, the variable should increase its 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 in 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., its value is first assigned) to where that value is *used.* We 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 of Figure 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, variable TryCount is defined by the statement "TryCount :=0" where variable TryCount is assigned the value of 0. We then describe 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 Figure 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, 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, variable ReqQos is used in the

output statement "Output connect(ReqQos)." Table 1 lists the variables defined and used in Figure 1.

Table 1. Variables defined and used at the nodes of the Data Flow Digraph of Figure 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

In tracing a variable X 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 in the tracing process. Thus, to trace such an association, we must construct the *define-clear-use path*[5] 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 Figure 1 and Table 1. Variable ReqQos and nodes a4 and o2 form a du-pair du(a4, o2, ReqQos) because it can be traced by the define-clear-use path [a4, d1, p2, o2], where 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 associations of variables and nodes form du-pairs. 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 an input node J (where a variable X is defined) to an output node K (where a variable Y is used), where X and Y can be either the same or different variables. Variables X and Y are input and output parameter values that are controlled and observed by the testing system respectively. Certainly, the value of variable X may not directly affect the value of variable Y, but it may affect indirectly. That is, X may affect another variable that in turn affects variable Y, and so on. For example, consider the path [i1, a4, d1, p2, o2] of Figure 1. At node i1, the EFSM receive 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

[5] the definition is extended from the define-clear path of [10][11].

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,...,p_{r-1}, p_r ,..., p_s (for variables X_1, X_2, ..., X_{r-1}, X_r, ...,X_s respectively) which starts from an input node J and ends at an output node K form an *input-output chain* (i.e., *io-chain)* io(J, X_1, K, X_s). The chain [p_1, p_2, .. .p_{r-1}, p_r ..., p_s] can be used to trace the data flow from an input node J where variable X_1 is input to an output node K where the value of 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 pr intersect, i.e., variable X_{r-1} affects X_r at the intersecting node (with the restriction that an action can has only one statement so that X_{r-1} cannot affect other variables). For example, [i1, a4] is a define-clear-use path for variable CONreq.qos which starts from an input node, and [a4, d1, p2, o2] is a define-clear-use path for variable ReqQos which ends at an output node. The two paths are connected into an io-chain io(i1, CONreq.qos; o2, ReqQos) which is [i1, a4, d1, 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.

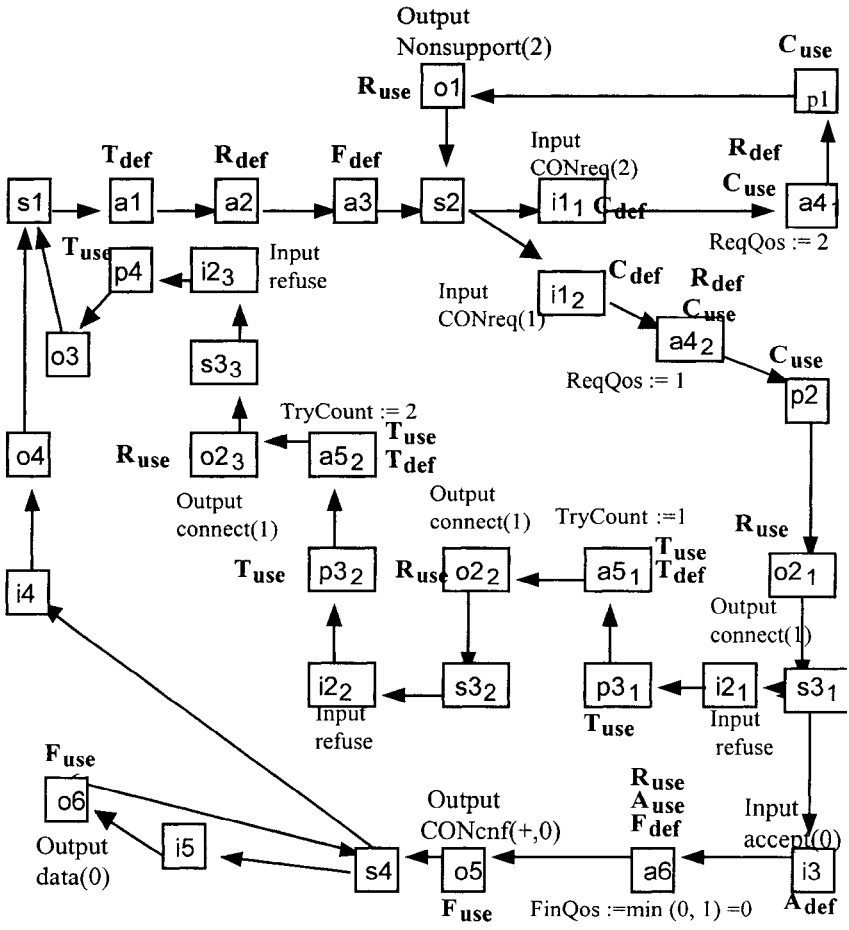
3. THE METHOD FOR THE FIRST CRITERION

In Section 2, we have described the All-Use and IO-df-chain criteria that are guidelines for selecting the test paths from the Data Flow Digraph. In this section, we will propose a method which automatically generates the executable test sequence that satisfies the first criterion. Our method involves four steps.

The first step involves the conversion of the Data Flow Digraph (Figure 1)
into the Behavior Machine Digraph (Figure 2) where executable paths can be directly generated. In general, a node J (Figure 1) is converted into nodes J_1, J_2, J_3, ... (Figure 2)[6] by embedding parameter values. If all parameter values are embedded, the Behavior Machine Digraph will be very large. Hence, we only embed parameter values that enable the removal of decision nodes. For example, consider node i1 (i.e., CONreq(qos)) of Figure 1. The parameter qos has values from 0 to 3, so that we can convert node "CONreq(qos)" into four nodes "CONreq(0)", "CONreq(1)", "CONreq(2)", "CONreq(3) ." However, because either "CONreq(0)" or "CONreq(1)" can make predicate p2 true (and either "CONreq(2)" or "CONreq(3)" can make predicate p1 true), we only create node "CONreq(1)" (i.e., node $i1_1$) and node

[6] Without loss of generality, notations J_1, J_2, J_3, ... of the Behavior Machine Digraph represent the nodes converted from the node J of the Data Flow Digraph in this paper.

"CONreq(2)" (i.e., node $i1_2$). Then we remove the decision node d1 from the digraph since no decisions are required there. Because every predicate in the digraph is true, any path of the digraph can be used to generate an executable test sequence. In Figure 2, the Behavior Machine is represented by fine lines and text, but it includes some bold characters that will be used in the next step for tracing the data flow. In general, a bold character "$\mathbf{X}_{\mathbf{def}}$" (or $\mathbf{X}_{\mathbf{use}}$) is put on nodes J_1, J_2, J_3, .. of Figure 2 if variable X is defined (used) at node J of Figure 1. For example, a bold character $\mathbf{C}_{\mathbf{def}}$ (an abbreviation of CONreq.qos_{def}) is put on nodes $i1_1$ and $i1_2$ of Figure 2, because variable CONreq.qos is defined at node i1 of Figure 1 (see Table 1.)



Remarks:
C: CONreq.qos **A**: accept.qos **T**:TryCount **R**: ReqQos **F**:FinQos
def (use): variable defined (used) d at that node

Figure 2. The Behavior Machine Digraph of the Data Flow Digraph of Figure 1 (only new statements other than that of Figure 1 are indicated)

The second step involves finding executable test paths of Figure 2 that can trace the du-pairs of Table 1. 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 of Figure 1 which starts from node J and ends at node K and does not redefine the variable in the intermediate nodes. In Figure 2, such an executable define-clear-use path corresponds to specific paths which start from nodes J_1, J_2,..., J_r,. . , J_s and end at nodes K_1 ,K_2 ,...,K_p,... ,K_q where the symbols "X_{def}" can only be seen in the starting nodes. These specific paths of Figure 2 can be constructed from shortest paths of Figure 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) of Table 1 where "CONreq" is defined at node i1 and used at node a4, we find the specific shortest paths which starts at nodes $i1_1$ and $i1_2$ and ends at nodes $a4_1$ and $a4_2$ of Figure 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 of Figure 2 for tracing the du-pairs of Table 1 are shown in Table 2.

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

Label	du-pairs2 of Table 1	Executable define-clear-use paths of Figure
1	du(i1,a4, CONreq.qos)	[$i1_1$, $a4_1$] or [$i1_2$, $a4_2$]
2	du(i1,p1, CONreq.qos)	[$i1_1$, $a4_1$, p1]
3	du(i1,p2, CONreq.qos)	[$i1_2$, $a4_2$, p2]
4	du(i3, a6, accept.qos)	[i3, $a6$]
5	du(a1, a5, TryCount)	[a 1 , a2, a3, s2, $i1_2$, $a4_2$, p2,$o2_1$, $s3_1$, $i2_1$, $p3_1$, $a5_1$]
6	du(a, p3, TryCount)	[a1, a2, a3, s2, $i1_2$, $a4_2$, p2,$o2_1$, $s3_1$,$i2_1$,$p3_1$]
7	du(a5, p3, TryCount)	[$a5_1$, $o2_2$, $s3_2$, $i2_2$, $p3_2$]
8	du(a5, p4, TryCount)	[$a5_2$, $o2_3$, , $s3_3$, $i2_3$, p4]
9	du(a4, o2, ReqQos)	[$a4_2$,p2,$o2_1$] or 2 p2, $o2_1$, $s3_1$, $i2_1$, $p3_1$, $a5_1$,$o2_2$] or [$a4_2$,p2,$o2_1$,$s3_1$ $i2_1$,$p3_1$,$a5_1$,$o2_2$,$s3_2$,$i2_2$, 2,$a5_2$,$o2_3$]
10	du(a4, a6, ReqQos)	2, p2, $o2_1$, $s3_1$, i3, a6]
11	du(a4, o1, ReqQos)	[$a4_1$, p1, o1]
12	du(a6, o5, FinQos)	[a6, o5]
13	du(a6, o6, FinQos)	[a6, o5, s4, i5, o6]

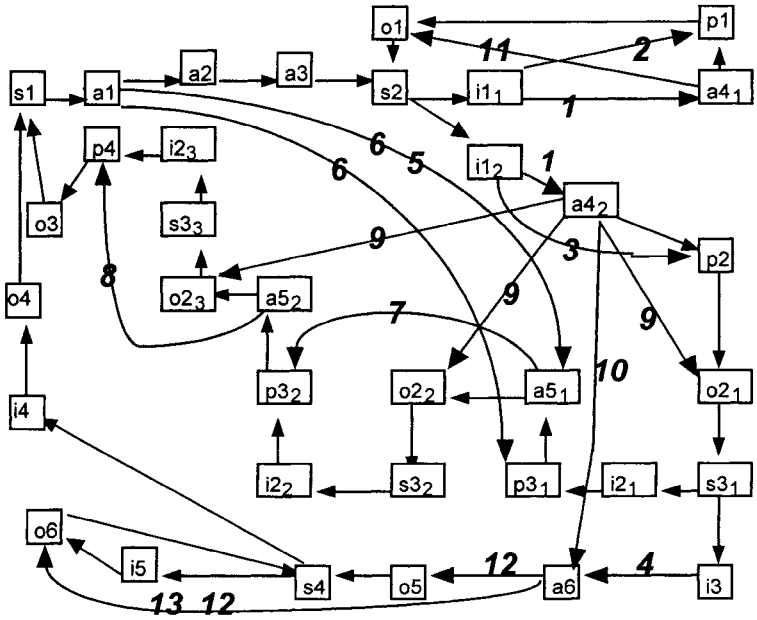


Figure 3. The SelectUse Digraph constructed from the Behavior Machine Digraph of Figure 2 by embedding the define-clear-use paths of Table 2 as bold edges

The third phase involves constructing a SelectUse Digraph of Figure 3 by embedding the executable test paths of Table 2 into the Behavior Machine Digraph of Figure 2. 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 put on the path to indicate the du-pair traced by the path. Cost is assigned to the edge according the number of inputs that it contribute to the final test sequence when that edge is traversed in the test path, because the length of a test sequence is decided by the number of inputs that it has. Thus, a fine edge is assigned a cost of 1 if it leaves an input node, and a bold edge is assigned a cost which is 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,$o2_1$, $s3_1$, $i2_1$, $p3_1$, $a5_1$] of Table 2 that traces the du-pair du(a1, a5, 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 al to $a5_1$. The bold edge is labeled with "5" and assigned a cost of 2. 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 of Table 2, and we put 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 properties that 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. 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 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 at 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, an Euler Tour of the Euler Digraph is an Selecting Chinese Postman Tour of the SelectIO Digraph. In the latter case, a tour will be used to connect these disconnected Euler Digraphs into an Euler Digraph, so that the Euler Tour algorithm can be applied. For Figure 3, the Lindo package [8] has solved the integer programming equations in less than one second and results in an Euler Digraph, resulting in the first case so that that the optimal tour is found. The Selecting Chinese Postman Tour of Figure 3: [s1, a1, a2, a3, s2, il_1, ***1***, $a4_1$, ***11***, o1, s2, $i1_1$, ***2,*** p1, o1, s2, $i1_2$, ***1*** , $a4_2$, ***10*** , a6, ***(13,12)*** , o6, s4, i4, o4, s1, a1, a2, a3, s2, $i1_2$, ***3***, p2, $o2_1$, $s3_1$, i3, ***4*** ,a6, ***12***, o5, s4, i4, o4, s1, al, a2, a3, s2, $i1_2$, ***1,*** $a4_2$, ***9,*** $o2_2$, $s3_2$, $i2_2$, $p3_2$, $a5_2$, $o2_3$, $s3_3$, $i2_3$, p4, o3, s1, a1, ***(6, 5)*** , $a5_1$, ***7*** , $p3_2$, $a5_2$, ***8*** , p4, o3, s1, a1, a2, a3, s2, $i1_2$, ***3***, p2, o21, s31, i3, ***4*** , a6, ***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, $i1_2$, $o2_1$, $i2_1$, $o2_2$, $i2_2$, $o2_3$, $i2_3$, 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 of Table 1.

4. THE METHOD FOR THE SECOND CRITERION

In Section 3, we have proposed a method to produce an executable test sequence which satisfies the first criterion by finding the Selecting Chinese

Postman Tour of the SelectUse Digraph. In this Section, we are going to extend the method to generate an executable test sequence which satisfies the second criterion, i.e., the IO-df-chain criterion which requires tracing each io-chain at least once. The extended method contains two steps.

The first step involves finding all io-chains of Figure 1. In the SelectUse Digraph of Figure 3, we consider input nodes J_1, J_2, J_3, ..., (converted from node J where variable X is defined) to output nodes K_1, K_2, K_3, ... (converted from node K where variable Y is used). As described in Section 2, an io-chain is constructed from a sequence of define-clear-use paths so as to connect an input node to an output node (but notice that a sequence of define-clear paths defined in [11] does not necessary form an input-output-chain). We call those bold edges which leave input nodes J_1, J_2, J_3, ..., that are the define-clear-use paths for variable X as the (X-J) edges, and those bold edges which enter output nodes K_1, K_2, K_3, ..., that are the define-clear-use paths for variable Y as the (Y-K) edges. A path which involves only bold edges is a bold path. In the SelectUse Digraph, the shortest bold path from these (X-J) edges to those (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). Those specific shortest paths of the SelectUse Digraph can be obtained from shortest paths of the SelectUse Digraph where fine edges are (temporarily) removed. For example, the bold edge (i.e., the define-clear-use paths [$i1_1$, ***1***, $a4_1$]) and another bold edge (i.e., [$a4_1$, ***11***, o_1]) compose an io-chain io(i1,CONreq.qos, o1, ReqQos) = [$i1_1$, ***1***, $a4_1$, $a4_1$, ***11***, o_1]. The complete io-chains for Figure 1 obtained from these specific shortest paths of Figure 3 are shown in Table 3, where each io-chain is given a label.

Table 3. The io-chains for the Data Flow Digraph of Figure 1

Label	input-output-chains (Figure 1)	Composed define-clear-use paths (Figure 3)	Corresponding Executable Paths (Figure 2)
a	io(i1,CONreq.qos, o1, ReqQos)	[$i1_1$, ***1***, $a4_1$, ***11***, o1]	[$i1_1$, $a4_1$, p1 o1]
b	io(i1,CONreq.qos, o2, ReqQos)	[$i1_2$, ***1***, $a4_2$, ***9***, $o2_1$]	[$i1_2$, $a4_2$, p2, $o2_1$]
		[$i1_2$, ***1***, $a4_2$, ***9***, $o2_2$]	[$i1_2$, $a4_2$, p2, $o2_1$, $s3_1$, $i2_1$, $p3_1$, a51, $o2_2$]
		[$i1_2$, ***1***, $a4_2$, ***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, CONreq.qos, o5, FinQos)	[$i1_2$, ***1***, $a4_2$, ***10***, a6, ***12***, 05]	[$i1_2$, $a4_2$, p2, $o2_1$, $s3_1$, i3, a6, o5]
d	io(i1, CONreq.qos, o6, FinQos)	[$i1_2$, ***1***, $a4_2$, ***10***, a6, ***13***, 06]	[$i1_2$, $a4_2$, p2, $o2_1$, $s3_1$, i3, a6, 05, s4, i5, o6]

The second step involves constructing a SelectIO Digraph of Figure 4 by embedding the executable test paths of Table 3 to the Behavior Machine Digraph of Figure 2. Generally, an executable test path listed in Table 3 which starts from an 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 Figure 4.) Costs are assigned to the edges of the SelectIO Digraph similar to the process described in the third step of Section 3 for assigning the cost to the edges of the SelectUse Digraph. A Selecting Chinese Postman Tour of the SelectIO digraph can be used to generate an minimum-cost executable test sequence that checks each io-chain. The tour is obtained using the algorithm described in the four step of Section 3. The Selecting Chinese Postman Tour of Figure 4: [s1, a1, a2, a3, s2, $i1_1$, ***a***, o1, s2, $i1_2$, ***b***, $o2_1$, $s3_1$, $i2_1$, $p3_1$, $a5_1$, $o2_2$, $s3_2$, $i2_2$, $p3_2$, $a5_2$, $o2_3$, $s3_3$, $i2_3$, p4, o3, s1, a1, a2, a3, s2, $i1_2$, ***(c, d)***, o6, s4, i4, s1] is used to generate the executable test sequence [$i1_1$, o1, $i1_2$, $o2_1$, $i2_1$, $o2_2$, $i2_2$, , $o2_3$, $i2_3$, o3, $i1_2$, $o2_1$, i3, o5, i5, o6, i4, o4] which traces all the io-chains of Table 3.

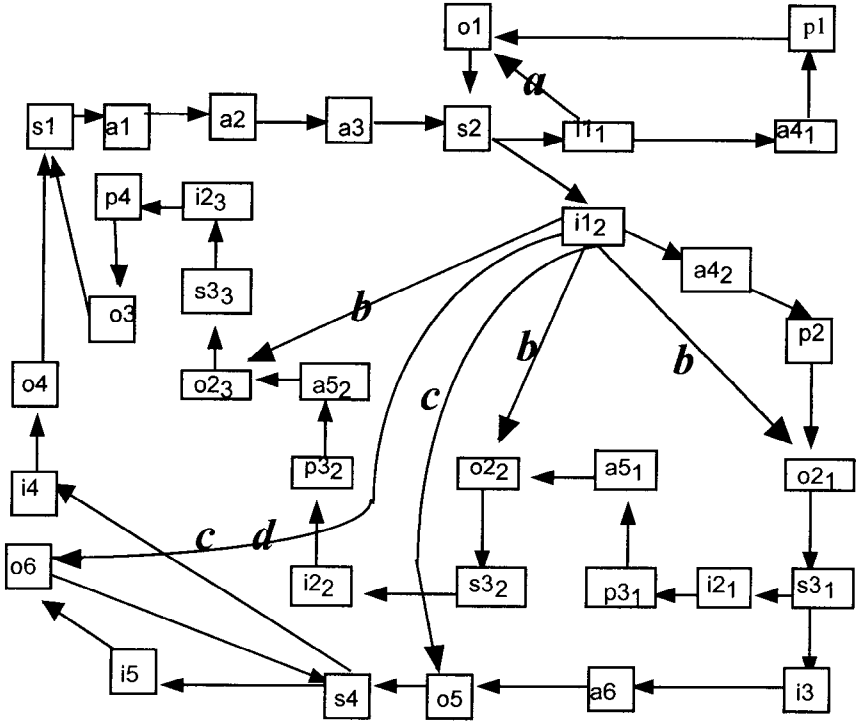


Figure 4. The SelectIO Digraph constructed from the Behavior Machine Digraph of Figure 2 by embedding the io-chains of Table 3 as bold edges

5. CONCLUSIONS

In this paper, we have proposed a method to automatically generate the executable test sequence from the protocol specification for verifying 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. The first phase of the Selecting Chinese Postman Algorithm involves solving a system of integer programming equations so as to find an augmentation. In our experiences of solving such equations [1][2], a linear programming version of the formulation always yield integer results. We want to check whether these equations satisfy a specific property so that the linear programming approach can be applied.

Our method of minimizing the test sequence length can be easily extended to minimizing the test sequence cost, by assigning cost to the edges of the digraph. And the method can be combined with the duplexE digraph method to generate a synchronizable and executable test sequence [3].

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.

[2] W. H. Chen, "Executable test sequence for the protocol control and data portions," *Proc. of IEEE Int'l Conference on Communications*, New Orleans, U. S. A., 2000.

[3] W. H. Chen and H. Ural, "Synchronizable test sequence based on multiple UIO sequences," *IEEE/ACM Trans. on Networking*, Vol. 3, No. 2, 1995.

[4] K. T. Cheng and A. S. Krishnakumer, "Automatic functional test generation using the extended finite state machine model," *Proc. IEEE Design Automation Conference*, 1993.

[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.

[6] E. Kwast, "Automatic test generation for protocol data aspects, " *Proc. IFIP Int'l Symp. on Protocol Specification, Testing, and Verification*, 1992.

[7] J. A. Mchugh, *Algorithmic Graph Theory,* Prentice-Hall, Englewood Cliffs, NJ, 1990.

[8] L. Schrage, *LINDO 5.0 User's 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, No. 5, 1987.

[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.

[12] C. J. Wang and M. T. Liu, "Generating test cases for EFSM with given fault models," *Proc. IEEE INFOCOM,* March 1993.