GENERATING TESTS FOR CONTROL PORTION OF SDL SPECIFICATIONS WITH SAVE[1]

Gang LUO, Anindya DAS, and Gregor v. BOCHMANN

Departement d'IRO, Universite de Montreal, C.P. 6128, Succ.A,
Montreal, P.Q., H3C 3J7, Canada E-mail:luo@iro.umontreal.ca, Fax: (514) 343-5834.

ABSTRACT
The signal SAVE construct is one of the features distinguishing SDL from conventional high-level specification and programming languages. On the other hand, this feature increases the difficulties of testing SDL-specified software. We present a testing approach consisting of the following three steps: SDL specifications are first abstracted into finite state machines with save constructs; next the resulting machines are transformed to equivalent finite state machines without save constructs; finally test cases are selected from the resulting finite state machines. In particular, we give a fault model for guiding test selection. We come up with an equivalent transformation method which works for all FSMs with save constructs for which equivalent FSMs exist. We also investigate the detection of faults of the input queue.

KEYWORDS: CCITT SDL, communication software, finite state machines, protocol conformance testing, protocol engineering, protocol verification, software testing, and SDL SAVE.

1. INTRODUCTION

At present, the three formal specification languages which have been accepted by international standards organizations for specifying communication software are SDL [1, 6], LOTOS [13] and ESTELLE [14]. Among them, SDL is the one which is most widely used in industrial applications [15, 1, 6, 17]. In particular, SDL is good for describing distributed computing systems and OSI [17, 1, 15]. Therefore, it is important to study the problem of testing SDL-specified software. It is noted that test selection methods developed for ESTELLE specifications (see for instance [16]) can also be adopted for SDL specifications, since both languages use an extended finite state machine model for their semantics. However, SDL contains a distinctive feature, the save constructs which increases its descriptive power considerably by providing a concise formalism for expressing the indeterminate order of arrivals of input signals. However, as pointed out in [1], the save construct was the first of

[1] This work was supported by the IDACOM-NSERC-CWARC Industrial Research Chair on Communication Protocols at the University of Montreal (Canada).

several divergences between SDL and CHILL -- a high level programming language recommended by CCITT -- that complicates the transformation from one language into the other, and its presence raises an added challenge to testing SDL-specified software. For example, in the test generation approach proposed in [2], it is assumed that there is no save construct in the simplified SDL specifications. In the area of testing SDL-specified software, it is a common practice to first transform SDL into FSMs (finite state machines) by neglecting parameters; testing is then conducted based on the transformed FSMs [3], since many effective testing methods are available for FSMs [10]. But, this practice has to either avoid using the save constructs in SDL specifications or neglect save constructs in the transformation. Otherwise, the transformed state machines, resulting from neglecting parameters, are FSMs with additional save constructs which we call *SDL-machines* , instead of FSMs. Therefore, methods are needed for testing SDL-machines.

We prove in this paper that in general SDL-machines cannot be modeled by equivalent FSMs (with input queues). However, we find that most SDL-machines obtained from practical SDL specifications can be modeled by FSMs which preserve the same input/output relationship. Therefore, it is possible to transform these SDL-machines into the equivalent FSMs in order to use the testing methods based on FSMs. Some initial efforts have been made to tackle this issue [4, 3, 5]. A formal method was presented in [4] and a similar framework was introduced informally through examples in [3]; however they did not consider equivalent transformation algorithms for the case where save constructs have several inputs, a case which is quite common. An equivalent transformation algorithm for the case of several inputs in the save constructs has been presented in [5], but the algorithm can only be applied to a subset of those SDL-machines for which the equivalent FSMs exist, not to all of them. In this paper, we generalize the approach introduced in [5] to obtain an equivalent transformation method which works for all the SDL-machines for which equivalent FSMs exist.

Based on the equivalent transformation, we present in this paper a test generation method. We give a fault model for guiding test selection which includes, in addition to transition faults, the faults related to save constructs and the input queue. We then generate test cases by the following steps: We first use the approach by [3] to abstract SDL specifications to SDL-machines by neglecting parameters. We next transform the SDL-machines to equivalent FSMs by using our algorithm. Finally we generate the test cases which consist of two parts: the first is derived from the equivalent FSMs by applying the test selection method for FSMs, and the second is derived for detecting the faults related to the input queue.

The rest of the paper is organized as follows. Section 2 is devoted to a brief introduction and formal notations for SDL-machines. Section 3 studies the equivalent transformation from SDL-machines to

FSMs. Section 4 handles the test case selection methods based on the results of Section 3, and analyzes the test coverage thus obtained.

2. BASIC CONCEPTS AND FAULT MODEL OF SDL-MACHINES

2.1. Basic concepts

(I) SDL-machines

We give in this section a brief introduction to what we call SDL-machines and to the save constructs [6, 7, 1]. An SDL-machine is a simplified SDL process which only has the following constructs: (a) "states", (b) "inputs", (c) "outputs", (d) "saves", (e) "transitions" and (f) an "input queue"; and it is actually a finite state machine with the extension of an FIFO (First In First Out) input queue and save constructs. Figure 1 lists a subset of SDL graphic symbols which are used to represent SDL-machines.

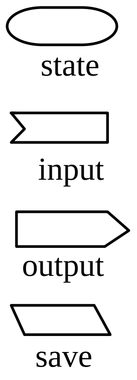

Figure 1. Illustration for SDL Graphic Symbols

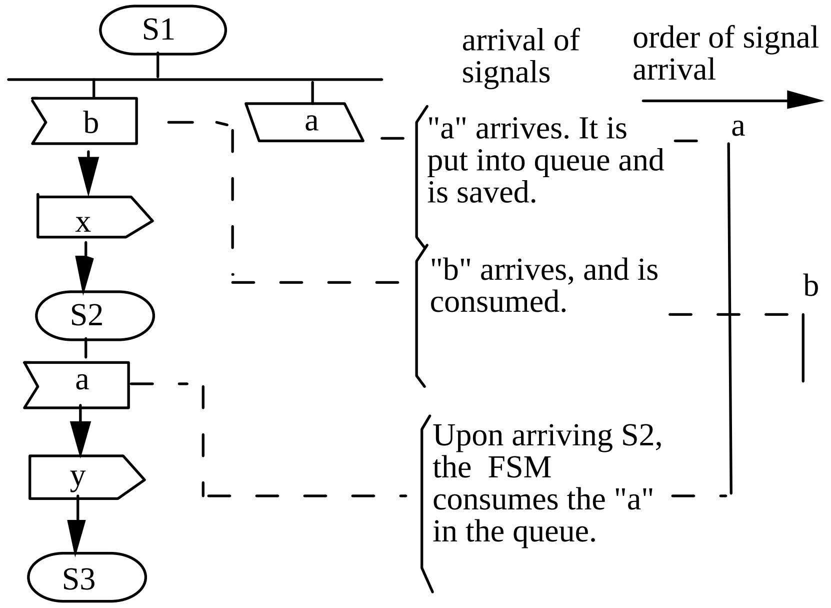
S1
b
a
x
S2
a
y
S3
arrival of signals
order of signal arrival
"a" arrives. It is put into queue and is saved.
"b" arrives, and is consumed.
Upon arriving S2, the FSM consumes the "a" in the queue.
a
b

Figure 2. Illustration for SDL signal SAVE

An SDL-machine consists of a FIFO queue and a finite number of states. Every arriving input is first placed in the input queue, and the following cases arise if in state S the input queue is not empty:

Case 1: All inputs in the queue are the inputs specified in the save construct of state S. In this case, the inputs are saved in the queue for future use; the SDL-machine is waiting for another input and it will not do anything further before another input is received.

Case 2: Among the inputs in the queue which are not specified in the save construct of state S, there is an input b which is the nearest to the front of the queue. In this case, the following two situations arise: (a) If the input b initiates a transition explicitly (i.e., the b is an input symbol attached to state S), the input b will be removed from the queue (it is consumed), the corresponding transition will be performed, and the SDL-machine will move to a next state. (b) If the input b does not initiate a transition explicitly, in this situation we have an explicitly unspecified reception. The input is discarded and an "implied transition" back to state S is said to have taken place.

For every state in an SDL-machine, the input symbol of every transition is different from the input symbols of the other transitions of the state, and also it is not a input symbol in the save construct of the state. Therefore, SDL-machines are deterministic state machines. These notions may be formalized as follows.

DEFINITION 1: An *SDL-machine* is a 7-tuple < K, I, O, saveset, nextstate, outp , s0> with an FIFO input queue where:
(1) K is a finite set of states.
(2) I is a finite set of inputs.
(3) O is a finite set of outputs.
(4) saveset is a function from K to powerset(I), saveset: K --> powerset(I);
where for a state S in K, saveset(S) is the set of inputs contained in the save construct of state S. saveset(S)* denotes the set of input sequences which only consist of the inputs in saveset(S).
(5) nextstate is a function from K x I --> K where for a state S in K, and an input "i" in I, nextstate(S, i) is the state to which the SDL-machine will transfer when it is in state S, input i is not in saveset(S) and is the foremost input in the queue.
(6) outp is a function from K x I --> O* where for a state S in K, and an input "i" in I, outp(S, i) is the output sequence which the SDL-machine will produce, if it is in state S, and if input i is not in saveset(S) and is the foremost input in the queue. This definition confirms that in SDL a transition may be associated with several outputs.
(7) s0 is the initial state in K.
[End of definition].

For the SDL-machine of Figure 2, we have the following:
K = {S1, S2, S3}, I = {a, b}, O = {x, y}, saveset(S1) = {a}, ...

For the sake of convenience, we introduce in the following several other notations for SDL-machines.
(1) @: K x I* --> K (I* is the transitive closure of I.)

@ is the extension of the function nextstate to sequences of inputs. Let S be a state and x an input sequence; S@x denotes a state reached from S by inputting x to the empty queue and by consuming inputs in x, such that in the resulting state the queue is either empty or only contains the inputs specified in the save construct of the resulting state.

(2) queue: K x I* --> I*

For S in K and x in I*, queue(S, x) is the content of the input queue after the SDL-machine with empty queue, being applied x, transfers from S to state S@x.

(3) op: K x I* --> O*

For a given state S and an input sequence x; op(Sgggg,x) denotes the output sequence produced by the transitions from S to S@x after inputting x to the empty queue in state S.

(4) out: K --> powerset(I)

For a given state A, out(A) is the set of input symbols attached to state A except for those in saveset(A).

(5) . : I* x I* --> I*, and O* x O* --> O*

We use the notation "." to represent the concatenation of two input sequences or two output sequences.

We use the example shown in Figure 2 to illustrate some of the above notations. For this case, saveset(S1)={a} and the saveset(S2) is an empty set. S1 @ a.b = S3 means that the SDL-machine in state "S1" consumes the input sequence "a.b" and transfers to state "S3". op(S1, a.b)=x.y means that the SDL-machine in state "S1" consumes the input sequence "a.b" with the sequence "x.y" sent as output; and out(S1)={b}. We also have queue(S1, a.a) = a.a and queue(S1, a.b) = "" (empty string).

(II) Save constructs

We illustrate the functioning of a save construct with the example shown in Figure 2. We assume that the SDL-machine is in state S1. If an input "a" arrives first and, it is kept in the queue because the symbol "a" appears in the save construct; if a "b" then arrives , it is consumed in state "S1", triggering a transition leading to state "S2" with an signal "x" sent as output. The "a" in the queue is consumed in state "S2" and the SDL-machine transfers to state "S3" with a signal "y" sent as output.

(III) Assumption for the input queue

There are usually two types of input queues in implementations for the state machines with input queues. The first assumes that if an input is consumed immediately after its arrival only a queue of zero length is needed; the other assumes that the length of the queue needed for the above case is at least one. We make the former assumption. For the example in Figure 2, if the length of the input queue in the implementation is 1, the input sequence a.b, being applied to state S1, does not cause any input loss under this assumption. However, the input sequence a.a.b will cause the loss of the second a, but the first a and b will not be lost.

(IV) The definition of equivalence

Before we generate test cases, we should answer the following question: what is the conformance relation to be satisfied between a specification and its corresponding implementation. Based on black-box-testing strategy where implementations are assumed to be black boxes, we answer this question by giving the following definition of the equivalence of SDL-machines in terms of input-output relations.

DEFINITION 2: An SDL-machine F1 is "*equivalent to*" an SDL-machine F2 if the following conditions hold:
(1) F1 and F2 have the same input set I, and the same output set O.
(2) Suppose that op and op' denote the op in F1 and F2, and that s0 and s0' denote the initial states in F1 and F2 respectively . Then, for every x in I*, op(s0, x) = op'(s0', x).
[end of definition].

The above equivalence definition serves as a guide to our test case generation method.

2.2. Fault Model for SDL-machines

We consider two categories of faults which can occur in SDL-machines. The first category includes the output and transfer faults as usually considered for FSMs: the output corresponding to a state transition is erroneous or there is a fault in the next state reached by a transition. The second category of faults is related to the save constructs and the input queue.

We now formalize the fault model for SDL-machines. Let SP be a specification and IUT its implementation. We assume that they have the same K,I, and O. The fault types are defined as follows:

(1) *Output fault:* We say that IUT has output faults if (i) IUT is not equivalent to SP, and (ii) SP can be obtained from IUT by modifying the outputs of several transitions in IUT.
(2) *Transfer fault:* We say that IUT has transfer faults if (i) IUT is not equivalent to SP, and (ii) SP can be obtained from IUT by modifying the ending states of several transitions in IUT.

(3) *Save fault:* We say that IUT has save faults if (i) IUT is not equivalent to SP, and (ii) SP can be obtained from IUT by modifying the contents (i.e., inputs) of several save constructs in IUT.

While an SDL specification assumes an unbounded queue, an implementation model has to limit the size of its queue to a prescribed "maximum length" [8] since no unbounded queue can be implemented in practical applications. We therefore consider the following additional fault.

(4) *Maximum length fault:* We say that IUT has a maximum length fault if the maximum length implemented is less than the one specified.

We will discuss the fault coverage of the test cases derived from our method on the basis of the fault model in Section 4.

3. TRANSFORMING SDL-MACHINES INTO FSMs

On the basis of the equivalence definition given before, we study in this section the method of transforming a given SDL-machine into an equivalent FSM with an input queue. The equivalent FSM with input queue is obtained by deleting all save constructs, by introducing additional states which do not have any save construct, and by incorporating additional transitions. We assume in the rest of the paper that the term "FSM" refers to the SDL-machine which does not have any save construct, and is in fact a finite state machine with an input queue.

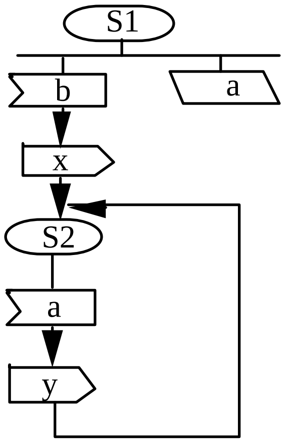

Figure 3. An example of an SDL-mchine without equivalent FSM

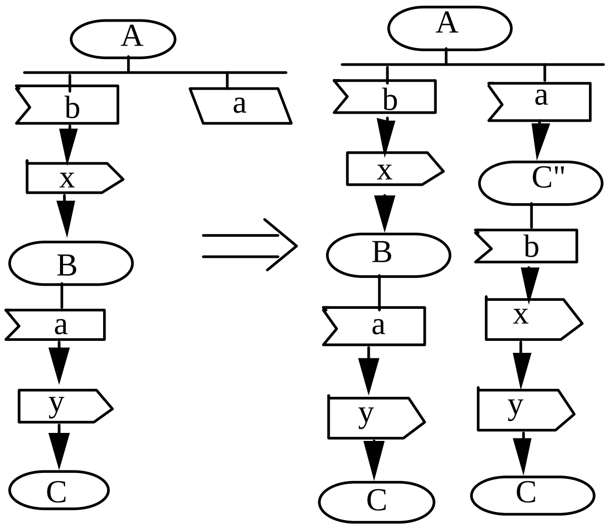

Figure 4. An example to illustrate the transformation

3.1. An example of an SDL-machine without an equivalent FSMs

We show in the following that, in the context of the equivalence definition given before, an SDL-machine, in general, cannot be modeled by an equivalent FSM although the SDL-machine resulting from most practical applications can be modeled by an equivalent FSM. An example of an SDL-machine for which there does not exist any equivalent FSM is illustrated in Figure 3. The following arguments show that the SDL-machine shown in Figure 3 does not have an equivalent FSM. From Figure 3, $op(S1,a^i.b)=x.y^i$ (note: a^i denotes a sequence of inputs "a" of length i and y^i denotes a sequence of outputs "y" of length i) in which the "i" may be any natural number. The output sequence "$x.y^i$" is produced only after input "b" is consumed; this means that the SDL-machine have the capacity to remember "a", "a.a", "a.a.a",......, an infinite number of input sequences. However, FSMs only have the capacity to remember a finite number of such input sequences; this leads to the following result.

THEOREM 1: There exist an SDL-machine which cannot be modeled by an equivalent FSM.

3.2. Equivalent transformation

We present in this section an algorithm which can transform an SDL-machine to an equivalent FSM. We also prove that the condition of applicability of the algorithm is a necessary and sufficient condition for the existence of an equivalent FSM. The following concept plays a key role in our algorithm.

DEFINITION 3: For S in K, x in saveset(S)*, the *p-sequence* (Prime input sequence) of x at state S is the input sequence constructed in the following manner:

Step 1: Let y initially be x.

Step 2: Stop if all inputs in x have been marked, and the resulting y becomes the p-sequence of x.

Otherwise, (i) find and mark the leftmost unmarked input "a" in y.

(ii) If $\forall z$□I*(if input a will be consumed by a transition t when x.z is applied to state S, then the transition t is an implied transition), then modify the input sequence y by eliminating the input a. Go to Step 2.

Furthermore, for S in K, x in saveset(S)*, we say x to be a p-sequence at S if the p-sequence of x is x itself.

[End of definition]

According to the above definition, for S in K and x in saveset(S)*, the p-sequence of x at S is unique. For the example shown in Figure 5, the p-sequence of a.a.b at state A is a.b. As another example, the set of all possible p-sequences at state A is {"", a, b, a.b, b.a, b.a.b}.

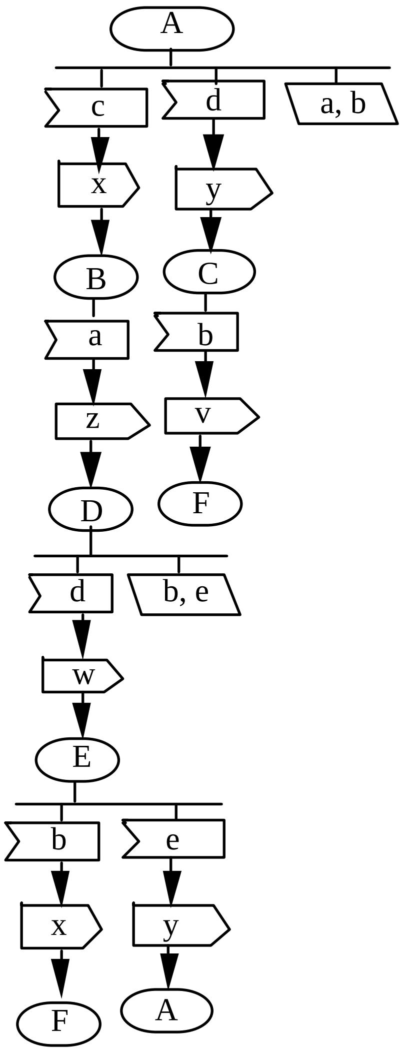

Figure 5. An example of SDL-machine

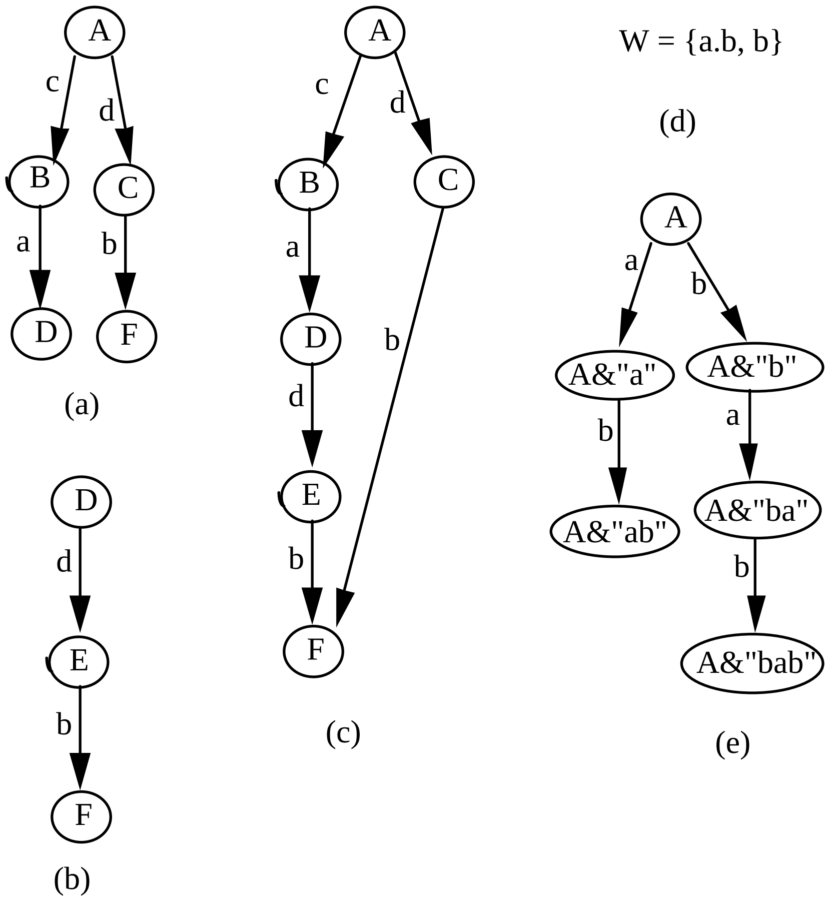

Figure 6. (a) the save-graph of A with respect to {a, b}; (b) the save-graph of D with respect to {b}; (c) the save-affected graph of A; (d) W set; (e) the save-corresponding-tree of A.

We now give an intuitive description of the equivalent transformation algorithm before presenting it more formally.

For a given state in an SDL-machine and an input, there are two cases as follows: the input is associated with a transition (or an implied transition) starting from the state and called a transition-associated input at the state; or the input is specified in the save construct of the state and called a save-associated input at the state. For a given state, a sequence consisting of only save-associated input at the state is said to be a save-associated input sequence for the state. A save construct at a given state in an SDL-machine is used to remember a save-associated input sequence received before a transition-associated input at that state is received. If the number of all the above possible save-associated input sequences, which can be saved in the queue, is finite, then, to avoid save constructs, intuitively we can build an equivalent SDL-machine from the original one by eliminating the save construct at the state and introducing a set of new states each of which remembers one of the above sequences. Unfortunately the number is obviously infinite since the sequences could be of arbitrary length. However, we find that not all inputs of the above save-associated input sequence contribute to possible output sequences since many inputs will be consumed later by implied transitions. Therefore, it is only necessary for the resulting equivalent FSM to remember every subsequence obtained from the above save-associated input sequences by eliminating all inputs which will be definitely discarded by implied transitions; these sequences are p-sequences for the corresponding states. Thus, the main part of our method is to find all the p-sequences for every state and use new states to remember the sequences in the resulting equivalent FSM if the number of possible subsequences is finite. The purpose of Algorithm 1 given below is to find all the p-sequences, Algorithm 2 builds the states to remember all the p-sequences, and Algorithm 3 adds the linking transitions. For the case where the number of the possible subsequences is infinite, we prove that the SDL-machine cannot be modeled by an equivalent FSM.

For the convenience of presenting algorithms, we require the following definitions. In order to use the terms in graph theory, we give a graph-theoretic definition for SDL-machines.

DEFINITION 4: An *SDL-graph* of an SDL-machine is a labeled directed graph where each edge corresponds to a transition of the SDL-machine with its label being a pair <Inputlabel, Outputlabel> which represents the "input " and "output sequence" of the transition; each node corresponds to a state, with the label of the node being a pair <S, saveset(S)> which represents the state name and the set saveset(S).
[End of definition]

According to the above definition, a transition has only one input, but may have an output sequence, as assumed in SDL. We often use in the following the term SDL-machine to refer to the SDL-graph when we talk about graphs, since they both have the same semantics.

For a given state S, and a set of inputs S-set, which is a subset of the inputs in the save construct of state S, a so-called *save-graph* of the given state S with respect to S-set is defined in the following such that: For an input sequence x which only consists of the inputs in S-set, and an input "a" in out(S), the input sequence x.a can only be consumed in the explicit transitions or implied transitions in the *save-graph* of the given state S with respect to the set of inputs S-set.

DEFINITION 5: A *save-graph* of a given state S with respect to a set of inputs S-set, is the largest directed subgraph of the SDL-graph F such that the edges of the subgraph consist of the following two parts: (i) All edges whose input labels are the inputs in S-set, and which can be reached through from one of the ending states of the outgoing transitions of state S, by a directed path in F with all inputs in the path belonging to S-set. (ii) All outgoing edges of S such that either the ending state of each edge, say Q, has an outgoing transition selected by (i), or the intersection of S-set and saveset(Q) is not empty and Q is not S.
[End of definition].

For the example shown in Figure 5, the save-graph of state A with respect to {a, b} is shown in Figure 6(a), and the save-graph of state D with respect to {b} is shown in Figure 6(b).

In order to present the equivalent transformation algorithm, Algorithm 3, we require the following Algorithms 1 and 2.

As mentioned in the beginning of this section, for every state, we need to find all the p-sequences. With this purpose in mind, Algorithm 1 given below constructs a so-called *save-affected-graph* of a given state, such that any one of maximal p-sequences can be obtained by deleting some inputs from an input sequence along a directed path in the graph; the inputs deleted are the inputs which are not the input symbols in the save construct of that state. It is obvious that the number of p-sequences is finite if the graph does not have any directed cycle.

ALGORITHM 1: Construction of the *save-affected-graph* of a given state.
Input: An SDL-machine F, and a given state S.
Output: A save-affected-graph of S.
Data structure: A stack of 2-tuples, < P, S-set >, where the first element is a state name, and the second is a set of inputs.
Step 1: Build a graph with only one node labelled S. Then put <S, saveset(S) > into the stack.
Step 2: Stop if the stack is empty. Otherwise:
(i) Pop a node < P, S-set > from the stack.
(ii) If all transitions of the save-graph of P with respect to S-set, are already in the resulting graph, then go to Step 2. Otherwise, find all transitions of the save-graph of P with respect to S-set, which are not yet in the resulting graph, and add the transitions and the adjacent states to the resulting graph.

(iii) For all leaf nodes in the save-graph (i.e. the states without outgoing edges), if, for such a leaf node Q, S-set ↔ saveset(Q) is not empty, then push < Q, S-set ↔ saveset(Q)> onto the stack. Go to Step 2.

[End of algorithm]

The save-affected-graph of S, resulting from Algorithm 1 is a subgraph of the given SDL-graph since all transitions and states in the save-affected-graph are obtained by selecting the transitions and states in the SDL-machine. The algorithm will terminate after a finite number of steps.

We require the following definition to present Algorithm 2.

DEFINITION 6: proj1 and proj2 are two functions from powerset(I*) to powerset(I*):

Let W□powerset(I*); we define:

proj1(W) = { z□ I* | ∃x□W ∃y□I* (z□"" & x = z.y)} where "" denotes the empty string.

proj2(W) = { z□ I* | ∃x□W (z□"" & "z is a sub-sequence of x, obtained by deleting zero or more inputs in x"}.

[End of definition].

For example, supposing W = {a.b, a.c, b}, we have proj1(W) = {a.b, a, a.c, b} and proj2(W) = {a.b, a, b, a.c, c}.

As mentioned in the beginning of this section, for a given state, after finding all the p-sequences, which can be done by using Algorithm 1, we need to introduce new states to remember the p-sequences and introduce the corresponding transitions between states such that each of the p-sequences leads the resulting machine from the given state to the state used for remembering that p-sequence. To achieve the above, the following algorithm constructs the states and the corresponding transitions which make up the so-called *save-corresponding-tree* of a given state . The transitions introduced in Algorithm 2 can only be labeled with the save-associated inputs.

ALGORITHM 2: Construction of the *save-corresponding-tree* of a given state S.

Input: An SDL-machine F, and a given state S.

Output: The save-corresponding-tree of a given state S.

Conditions of applicability: The save-affected-graph of state S does not contain any directed cycle which has at least one input in saveset(S).

Step 1: Construct an input sequence set W:

W = { x□I* | y is an input sequence along a maximal directed path in the save-affected-graph such that the path starts from the root and has at least one input in saveset(S), and x is derived by eliminating all inputs of y which are not in saveset(S) }. The save-affected-graphs are obtained using Algorithm 1.

Step 2: Build a tree containing only one node labeled S&"". (Note: "" stands for the empty string)

Step 3: Stop if all leaves of the resulting tree have been marked. Otherwise, find and mark in the resulting tree a leaf node labeled S&"x" where the node has not yet been marked. For all b in saveset(S), if proj1(W) ↔ proj2({x}) □ proj1(W) ↔ proj2({x.b}), create a child node of S&"x" with label S&"x.b". Goto Step 3.

[End of algorithm]

The algorithm will terminate after a finite number of steps, since we can find an upper bound such that we have proj1(W) ↔ proj2({x}) = proj1(W) ↔ proj2({x.b}) = proj1(W) for any b if the length of x is longer than the bound. The save-corresponding-graph of state S, resulting from Algorithm 2, is deterministic if it is considered as an FSM.

Suppose that, for an SDL-machine at a given state, say S1, there is a save-associated input sequence in the queue, which can be represented by a new state, say A, created by using Algorithm 2. If a transition-associated input is received, then the machine may transfer to another state, say S2, with the queue containing the remaining inputs, passing through several transitions, and generating an output sequence. Another new state, say B, created by using Algorithm 2, can represent the situation that the machine is at the state S2 with the queue containing the remaining inputs. In the resulting equivalent FSM, we require a transition from the first new state, state A, to the second new state, state B, with the input being the above transition-associated input and the output the above output sequence. Algorithm 3 given below is used for creating all such transitions.

ALGORITHM 3: Equivalent transformation of SDL-machines to avoid save constructs.

Input: An SDL-machine F.

Output: An equivalent FSM.

Conditions of applicability: For every state S in K, the save-affected-graph of S does not contain any directed cycle which has at least one input in saveset(S).

Step 1: For every S in K, if the saveset(S) is not empty, build the save-corresponding-tree using Algorithm 2. Rewrite all the tree such that all nodes, labels and edges are changed into the states, inputs and transitions in SDL-graphic symbols.

Step 2: If all save-corresponding-trees have been marked, stop. Otherwise, find and mark such an unmarked tree and do the following: Suppose that the label of the root of the tree is S&"". For all nodes S&"x" in the tree except for the root, and for all "a" in out(S), create a directed edge from node S&"x" to the node Q with input a and output op(S, x.a) where Q is decided as follows:

Let S1 = S@x.a, z = queue(S, x.a) and y be the p-sequence of z at state S1; then Q is node S1&"y", where y may be an empty string.

Step 3: Rename all S&"" by S in the resulting graph. As a result, the equivalent FSM consists of the resulting graph and the graph obtained from F by deleting all save constructs.

[End of algorithm].

The algorithm will terminate after a finite number of steps. It is easy to see that the machine obtained by applying Algorithm 3 is a deterministic FSM. The validity of the algorithm is given by the following theorem.

THEOREM 2: Algorithm 3 transforms SDL-machines into the equivalent FSMs under the conditions of applicability.

Proof: See Appendix.

The following theorem shows that the applicability condition of the algorithm is a necessary and sufficient condition for the existence of an equivalent FSM foe a given SDL-machine.

THEOREM 3: For a given SDL-machine, there exists an equivalent FSM iff the condition of applicability of Algorithm 3 is satisfied.

Proof: See Appendix.

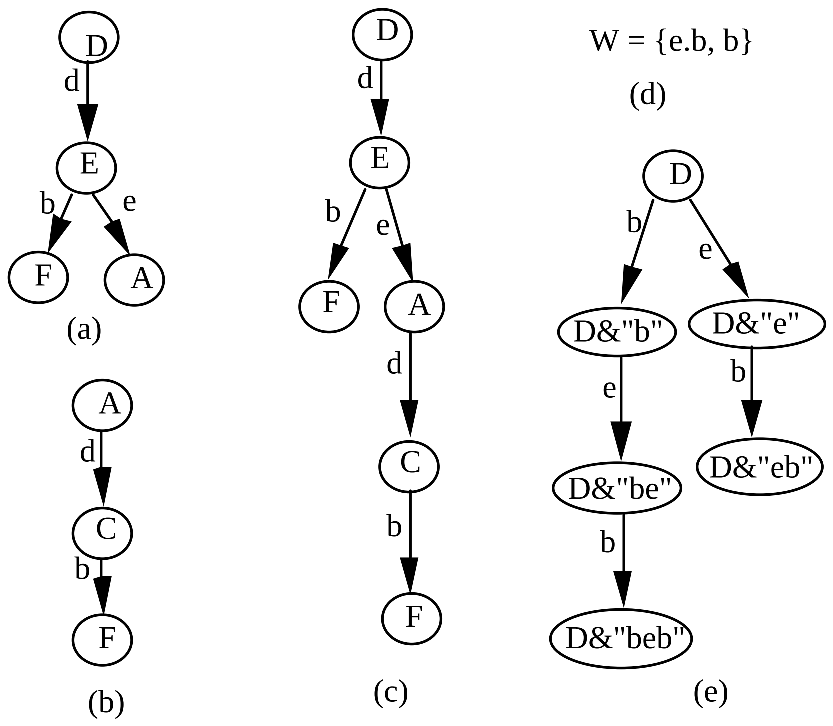

Figure 7. (a) the save-graph of D with respect to {b, e}; (b) the save-graph of A with respect to {b}; (c) the save-affected graph of D; (d) W set; (e) the save-corresponding-tree of D.

In the following, we illustrate the working of the algorithms for equivalent transformation by using the example shown in Figure 5. The transformation consists of the following stages:

(I) Constructing save-affected-graphs by Algorithm 1:

In this example, only states A and D have save constructs. For state A, we first construct the save-graph of state A with respect to {a, b}, and the save-graph of state D with respect to {b}, which are shown in Figures 6(a) and 6(b). We then construct the save-affected-graph of A which is shown in Figure 6(c). For state D, we have similar results, which are shown in Figure 7.

(II) Constructing save-corresponding-trees by Algorithm 2:

For state A, we first derive the W set in the step 1 of Algorithm 2, which is shown in Figure 6(d). We then produce the save-corresponding-tree of A, shown in Figure 6(e). For state D, we have similar results, as shown in Figure 7.

(III) Constructing the equivalent FSM by Algorithm 3:

From the earlier results, we construct the equivalent FSM as shown in Figure 8.

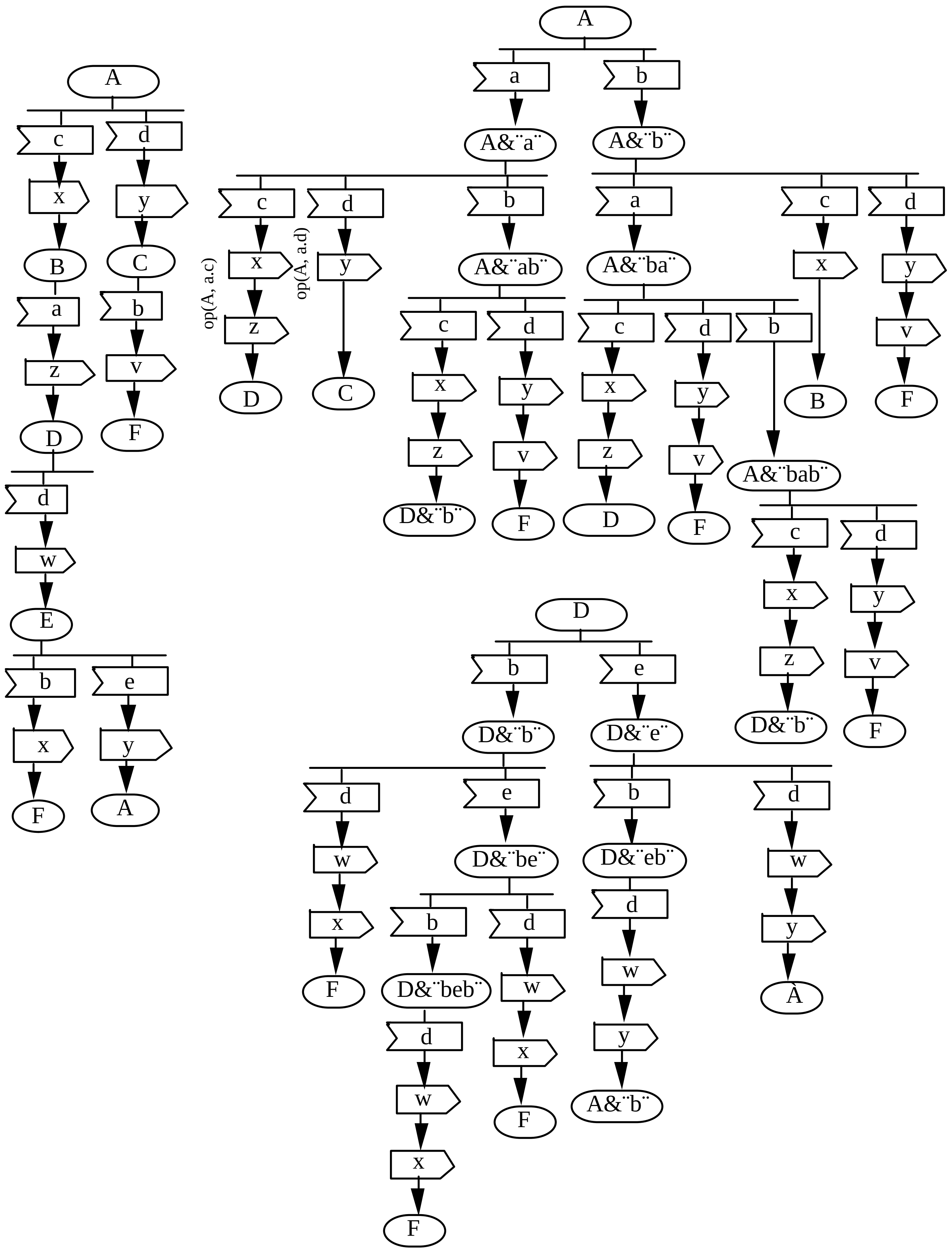

Figure 8. The equivalent FSM

4. TEST DESIGN

Guided by the fault model presented in Section 2, we give in this section a method for test selection from SDL-machines, which is based on our equivalent transformation algorithms. Our method considers not only the usual output faults and transfer faults for FSMs, but also the save faults and the maximum length faults which are specific to SDL-machines. We also analyze the test coverage of the test cases derived by our method under the fault model.

Test generation for an SDL-machine :
Step 1: Transform the given SDL-machine to an FSM using Algorithm 3.
Step 2: Generate test cases from the transformed FSM using one of the test suite development methods for finite state machines, such as the W-method [9], the Wp-method [10], UIO-method [11] or Transition tour [12].
Step 3: If the implementation is supposed to limit the maximum size of the input queue to *U*, then find an input sequence p.x.y which satisfies the following:
∃ A □K, x □ saveset(A)*, the length of x being *U*, p,y □ I∗, s0@p = A, and queue(s0,p) = empty such that :
∀ z □ proj1({x}) (if z□x, then op(A, x.y) □ op(A, z.y))
We call p.x.y a boundary test case.
[End of algorithm].

In the step 3 of the above algorithm, the input sequence p is the preamble leading to state A from the initial state s0.

Consider the example shown in Figure 5. Suppose the maximum queue length is 2. Let p = empty string, x = a.b, and y = c.d. Then a.b.c.d is the boundary test case. If the length of the queue is less than 2, say 1, in the implementation, the "b" will be lost after receiving the input sequence a.b and the output sequence is equal to op(A, a.c.d) which is not equal to op(A, a.b.c.d), as shown in Figure 5.

We analyze in the following the fault coverage under the fault model presented in Section 2, which means that we assume that no faults other than those in the fault model can occur.

The fault coverage of the test suite:
Case 1: The condition of applicability is satisfied.

If the methods used in the above Step 2 can ensure the complete fault coverage for FSMs under the fault model which only assumes the output fault and the transfer fault, the test suite can detect any fault specified in the fault model of Section 2. This is because save faults can be modeled as output faults and transfer faults in the transformed equivalent FSMs. The test case generated in the Step 3 can detect the maximum length fault. This means a complete coverage.

Case 2: The condition of applicability is not satisfied.

In this case, test case selection is based on the transformed FSM which is an approximation of the original SDL-machine. Therefore complete fault coverage cannot be guaranteed.

We notice that most SDL-machines derived from practical SDL specifications have equivalent FSMs. Therefore, our approach results in good fault coverage for most practical applications. For situations where the condition of applicability of Algorithm 3 does not hold, an approach to test generation was given in [5] although the equivalence relation between specifications and implementations cannot be fully guaranteed.

5. CONCLUSION

We presented in this paper a test generation method for SDL specifications which is based on the equivalent transformation algorithm. The most important step of our method is to transform a given SDL-machine into an equivalent FSM. We come up with the equivalent transformation algorithm which works for all SDL-machines for which the equivalent FSMs exist. The save constructs in SDL also cause similar problems in the area of verification and trace analysis for SDL-specified software. Therefore, the equivalent transformation method could be useful in those areas as well. Furthermore, if the outputs are neglected and some final states are specified, SDL-machines can be used to accept a language. In this case, it is a new type of automata and could be of interest in the area of automata theory.

APPENDIX: Validation of the Method

The outline of proof of Theorem 2:

We can only present the outline of the proof for Theorem 2 because of the length limitation of the paper.

For convenience, we assume the following:

(1) F is an SDL-machine which satisfies the condition of applicability of Algorithm 3. (2) F' is the resulting machine obtained from F using Algorithm 1, 2 and 3. (3) In contrast to the @ and op functions for F, @' and op' refer to the corresponding functions for F' respectively.

We first prove the following four lemmas:

LEMMA 1: For S□Kand x□saveset(S)*, suppose that the p-sequence of x is z, then for all y□I*, op(S, x.y) = op(S, z.y) and S@x.y = S@z.y. *(It has been proved using Induction on the length of x).*

LEMMA 2: For S□K, x□saveset(S)*, the p-sequence of x can be obtained as follows:
Apply x to S in the save-corresponding-tree to find a directed path from the root S to the node such that every input of x is eventualy consumed, by considering the tree as an SDL-machine; the input sequence along the path is the p-sequence of x at S. (In particular, when z is the p-sequence of x, we have S@'x = S@'z.).*(It has been proved using Induction on the length of x).*

LEMMA 3: Let S□K, a□I, x□saveset(S)* and x be p-sequence at S. The following statements hold:
(a) op(S, x.a) = op'(S, x.a);
(b) S@'x.a = (S@x.a)@'queue(S, x.a);
(c) op'(S@x.a, queue(S,x.a).z) = op'((S@x.a)@'queue(S, x.a), z) for z□I*.
(It has been proved using Lemma 2).

LEMMA 4: For S□K, x□I*, we have op(S, x) = op'(S, x). (It has been proved using Induction on the length of x and using Lemmas 1, 2 and 3)
Then, Theorem2 can be proved from Lemma 4.
[End of proof].

The computation complexity of Algorithm 3 is not very high in most practical cases according to our experiences, although it is very high in the worst cases.

Proof of Theorem 3: *(I. Sufficiency):* Theorem 2 proves that the condition is sufficient.
(II. Necessity): To prove that the condition is also necessary.
Assume the contrary that there exists Q in K such that the save-affected-graph of Q contains a directed cycle which has at least one input in saveset(Q). In this case, we find the shortest path from Q to the cycle; and we assume that

(i) x1 is the input sequence along the path,
(ii) S is the ending state of the path,
(iii) x2 is the input sequence along the cycle from S to itself,
(iv) y1 is obtained by eliminating all the inputs of x1 which are not in saveset(Q)*,
(v) z1 is obtained by eliminating all the inputs of x1 which are saveset(Q)*,
(vi) y2 is obtained by eliminating all the inputs of x2 which are not in saveset(Q)*,
(vii) z2 is obtained by eliminating all the inputs of x2 which are saveset(Q)*.

Then, we have op(S, $x1^{n}.x2^{n}$) = op(S, $y1.z1.y2^{n}.z2^{n}$) = op(S, $y1.y2^{n}.z1.z2^{n}$), (Note: n>0, x^{1}=x, $x^{n}=x.x^{n-1}$). When $y1.y2^{n}.z1.z2^{n}$ is applied to state S, the inputs of $y2^{n}$ can only be consumed

explicitly after z1 is consumed. Therefore, the SDL-machine has to have the capacity to remember all possible $y2^n$ where n = 1, 2, 3, ... FSMs only have the capacity to remenber a finite number of such sequences. Thus, the condition is necessary.
[End of proof].

REFERENCES:

[1] Roberto Saracco, J.R.W. Smith and Rick Reed, Telecommunications Systems Engineering Using SDL, North-Holland, 1989.
[2] Dieter Hogrefe, "Automatic generation of test cases from SDL specifications", SDL Newsletter, No.12, June 1988.
[3] Anne Bourguet-Rouger & Pierre Combes, "Exhaustive Validation and Test Generation in Elivis", SDL Forum'89: The Language at Work, North-Holland, 1989.
[4] Gang Luo & Junliang Chen, " Investigation & Testing for SDL SAVE Function" , Journal of Beijing Univ. of Posts & Telecommunications, Vol.12, No.4, December, 1989.
[5] Gang Luo, Anindya Das and Gregor von Bochmann, "Test Selection Based on SDL specification with Save", SDL'91 Evolving Methods, Proceedings of 5th SDL Forum, Ed. by O. Faergemand and R. Reed, North-Holland, 1991, pp.313--324.
[6] F. Belina and D. Hogrefe, "The CCITT-Specification and Description Language SDL", Computer Networks and ISDN Systems, Vol. 16, pp.311-341, 1989.
[7] R.Saracco & P.A.J.Tilanus, "CCITT SDL: Overview of Language and Its Application", Computer Networks and ISDN System, Vol.13, No.2, 1987, PP.65-74.
[8] G.v. Bochmann, A. Das, R. Dssouli, M.Dubuc, A.Ghedamsi, and G.Luo, "Fault Model in Testing", Proceedings of International Workshop on Protocol Testing Systems, Oct. 15-17th, 1991, the Netherlands.
[9] T.S.Chow, "Testing Software Design Modeled by Finite-State Machines, IEEE Trans. on Software Eng., Vol. SE-4, No.3, 1978.
[10] S.Fujiwara, G.v. Bochmann,F.Khendek,M.Amalou & A.Ghedamsi, "Test Selection Based on Finite State Models", IEEE Trans. on Software Engineering, Vol SE-17, No.6, June, 1991, pp.591-603.
[11] K.Sabnani & A.T.Dahbura, "A New Technique for Generating Protocol Tests", ACM Computer Communication Review, Vol.15, No.4, 1985, pp.36-43.
[12] B.Sarikaya & G.v.Bochmann, "Some Experiences with Test Sequence Generation for Protocols", Protocol Specifications, Testing, and Verification II, 1982.
[13] T. Bolognesi, Ed Brinksma, "Introduction to the ISO Specification Language LOTOS", Computer Network and ISDN Systems, Vol.14 (1) (1987), pp.25-59.
[14] S. Budkowski, P. Dembinski, "An introduction to Estelle: a specification language for distributed systems", Computer Network and ISDN Systems, Vol.14 (1) (1987), pp.3-23.
[15] O. Faergemand and R. Reed, (Ed.) SDL'91 Evolving Methods (Proceedings of 5th SDL Forum), North-Holland, 1991.
[16] B. Sarikaya, G.v. Bochmann, and E. Cerny, "A Test Design Methodology for Protocol Testing", IEEE Transactions on Software Engineering, Vol.13, No.9, Sept.. 1987, pp.989-999.
[17] F. Belina, D. Hogrefe and S. Trigila, "Modelling OSI in SDL", in K.Turner (Ed.), Formal Description Techniques, North-Holland, Amsterdam, 1988.