TEST SELECTION BASED ON SDL SPECIFICATIONS WITH SAVE

Gang LUO, Anindya DAS and Gregor v. BOCHMANN

Department d'IRO, Universite de Montreal, C.P. 6128,Succ.A, Montreal, P.Q., H3C 3J7, Canada. E-mail:luo@iro.umontreal.ca, Fax: (514) 343-2155 .

Abstract

The signal SAVE function is one of the characteristics distinguishing SDL from conventional high-level specification and programming languages. However, this feature increases the difficulties of testing SDL-specified software. This paper proposes a method for developing tests for system testing based on SDL specifications including the SAVE construct. It also investigates the effects of the input queue of SDL.

1. INTRODUCTION

During the development of SDL, the first feature added to SDL which considerably increased the difficulty of transforming SDL to CHILL was the SAVE construct[1]. However, the SAVE function increases SDL's descriptive power considerably by providing a concise formalism for expressing the indeterminate order of arrivals of input signals. Its presence raises a challenge in testing SDL-specified software. Some initial efforts have been made to tackle this issue [2,3]; a formal method was proposed in [2] and a similar framework was introduced informally through examples in [3]. However they did not address the case where the SAVE construct has several SIGNALs, a case which is quite common.

This paper investigates software testing based on SDL specifications when SAVE constructs contain several signals. Our approach is to transform an SDL description containing SAVE to an equivalent SDL description without SAVE which preserves the same relationship between input signal sequences and output signal sequences. The testing methods for the finite state machine can then be applied [4,5,6,7]. In the case of an SDL description which does not have an equivalent finite state machine (FSM) without SAVE, an alternate approach is proposed.

Our approach assumes that the SDL description is a FSM with the SAVE extension. Such a description can be obtained from a general SDL specification in the following fashion. The variable extension can be eliminated by transforming conditions which cause branches at the DECISION construct; the combinations of inputs and conditions can be used to create a FSM with new inputs being the combination of conditions and original inputs. The details can be obtained from [3]. By ignoring parameters and other variables, we obtain a finite state machine containing SAVEs and an input queue, which we call an "*SDL-machine*".

The rest of the paper is organized as follows. Section 2 is devoted to the fault model and gives a brief introduction to the *SDL-machine* formalism. Section 3 investigates the relations between *SDL-machine*s and FSMs in order to adopt the testing methods for FSMs to test *SDL-machine*s. We propose an algorithm to transform an *SDL-machine* to an equivalent FSM

which preserves the input/output relation. For the *SDL-machine* which cannot be transformed to an equivalent FSM leaving the input/output relation unchanged, another algorithm is given to transform it to a FSM which approximates the original *SDL-machine*. Section 4 handles the test case selection methods based on the results of section 3, and analyzes the test coverage thus obtained.

2. SDL SIGNAL SAVE AND A FAULT MODEL

A brief introduction to SDL and its signal SAVE construct[1,8,9] is given in this section, and a fault model for *SDL-machine* is proposed.

The *SDL-machine* consists of a FIFO queue and a finite control. Arriving inputs are placed in the input queue. The following cases arise if in state A the input b is the first element of the input queue:

Case 1: The input b can initiate a transition. The input b is removed from the queue (it is consumed) and the corresponding transition is performed.

Case 2: The input is a SAVE signal for state A. The input is stored in the queue for future use.

Case 3: Neither of the above cases holds. In this case we have an explicitly unspecified reception. The input is discarded and an "implied transition" to state A is said to have taken place.

There are two types of input queues, one 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 above case is at least one. We make the latter assumption.

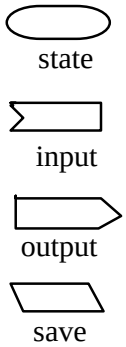

Figure 1.
Illustration for SDL
Graphic Symbols

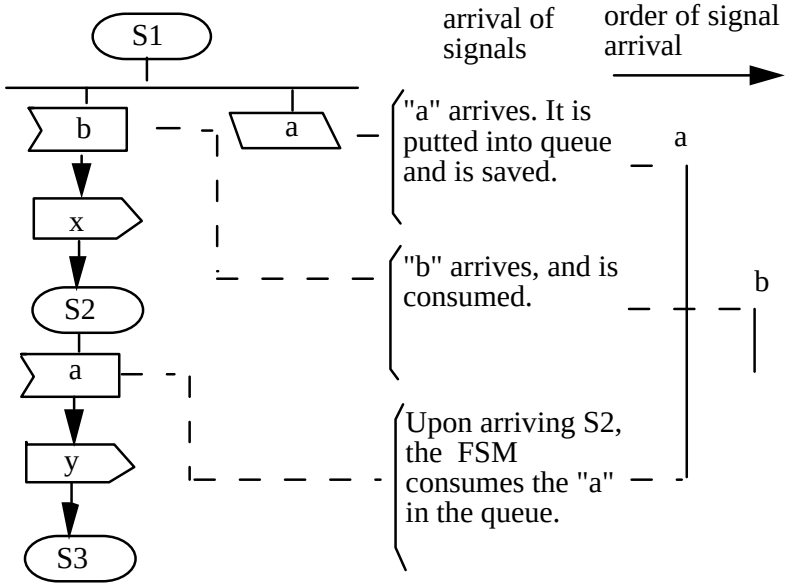

Figure 2. Illustration for SDL signal SAVE

2.1. Notation

Figure 1 shows graphic representations of some constructs used in an *SDL-machine*. Figure 2 illustrates the SAVE function. For the example shown in Figure 2, we assume that the *SDL-machine* is in state S1, a signal "a" arrives first and is kept in the queue because the symbol "a" appears in the SAVE; then "b" arrives , is consumed in state "S1", triggering a transition leading to state "S2" with a 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.

For the sake of convenience, several notations are introduced. Let K be the set of states, I the set of input signals, and O the set of output signals. Several functions are defined as follows:

(1) @: K x I* --> K (note: I* is the transitive closure of I.)

Let A be a state and x an input sequence; we write <A>@[x] for a state reached from A by inputting x. For the example shown in Figure 2, <S1> @ [a.b] = <S3> means that the *SDL-machine* in state "S1" consumes the input signal sequence "a.b" and transfers to state "S3".

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

Let A be a state and x an input sequence; we write op(A,x) for the output sequence produced by the transitions from A to <A>@[x] after inputting x in state A. For the example shown in Figure 2, op(S1, a.b)=x.y means that the *SDL-machine* in state "S1" consumes the input signal sequence "a.b" with the signal sequence "x.y" sent as output.

(3) save: K --> powerset(I)

Given a state A, save(A) is the set of input signals which are contained in the SAVE constructs associated with state A. For the example shown in Figure 2, save(S1)={a}. The save(S1) contains all signals attached to the SAVE constructs of state "S1". No SAVE construct is attached to state "S2"; the save(S2) is an empty set.

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

Given a state A, out(A) is the set of input signals attached to state A. For the example shown in Figure 2, out(S1)={b}.

2.2. Fault Model for SDL-machines

We consider two categories of faults which can occur in *SDL-machine*s. The first category corresponds to the usual output and transfer faults considered in 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 construct. The faults related to the SAVE construct for a given state are: (a) The SAVE inputs associated with the given state in the implementation under test (IUT) do not correspond to the SAVE inputs in the specification; (b) for a fixed N, the SAVE construct is not correctly implemented for all valid input sequences of length at most N.

We now formalize the fault model for *SDL-machine*s. 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. If there exists A belonging to K, "a" belonging to I, such that the op(A, a) of SP is not equal to the op(A,a) of IUT; we say that the IUT has an output fault.

(2) Transfer fault. If there exists A belonging to K, "a" belonging to I, such that <A>@[a] of SP is not equal to <A>@[a] of IUT; we say that the IUT has a transfer fault.

(3) Save input fault. If there exists a A belonging to K such that the save(A) of SP is not equal to the save(A) of IUT; we say that the IUT has a save input fault.

(4) Multiple saved inputs fault (up to N). If there exists A belonging to K, "a" belonging to I, x belonging to (save(A))*, and the length of x is less than N such that <A>@[x.a] of SP is not equal to <A>@[x.a] of IUT , or op(A,x.a) of SP is not equal to op(A,x.a) of IUT; we say that the IUT has a multiple saved inputs fault (up to N).

3. TRANSFORMING SDL-MACHINES INTO FSMs

In this section, we consider the transformation of a given *SDL-machine* into an equivalent FSM. We assume that the *SDL-machine* and the transformed FSM have infinite input queues.

3.1. An Example of an SDL-machine without an equivalent FSM

In general the *SDL-machine* is not equivalent to a FSM in that it may have an infinite number of states. In most cases of practical application, however, there is an equivalent FSM. Figure 3 shows an example for which we cannot find an equivalent FSM without SAVE which preserves the relation between the input and output signal sequences.

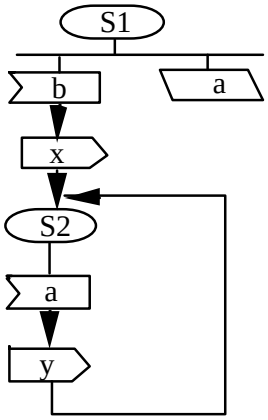

Figure 3. An example of an SDL-mchine without equivalent FSM

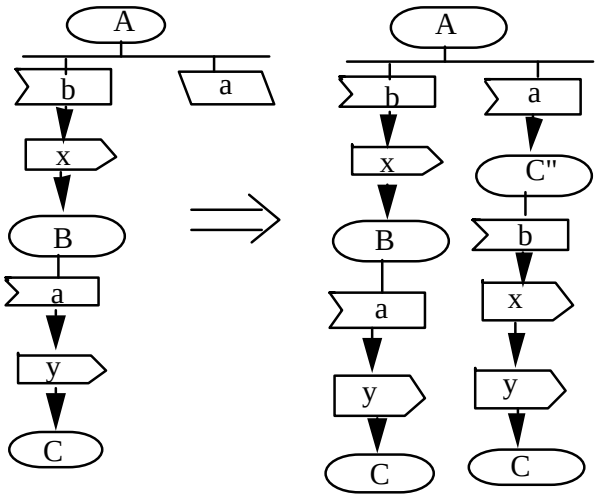

Figure 4. An example to illustrate the transformation

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 signals "a" of length i and y**i denotes a sequence of signals "y" of length i) in which the "i" may be any natural number. The output signal sequence "x.y**i" can come out only after the "b" is input; therefore the FSM has to have the capacity to memorize "a", "a.a","a.a.a",....., an infinite number of sequences. This is contrary to the definition that the FSM has only a finite number of states.

3.2. Equivalent Transformation

In the following, we describe an algorithm for transforming a given *SDL-machine* into an

equivalent FSM. First a simple example is given to illustrate the idea of the transformation, then an algorithm for the transformation is presented. The algorithm is illustrated by an example.

The example shown in Figure 4 illustrates the key concepts of the transformation although the comprehensive method is more complex. In order to present the algorithm, we require the following definitions.

DEFINITION: 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 signal" and "output signal sequence" of the transition; each node corresponds to a state, with the label of the node being a pair <Stateid, save(Stateid)> which represents the state name and the set save(Stateid).

DEFINITION: A *Save-Subgraph of state A* is the largest directed subgraph of the *SDL-graph* in which every edge has an input label which belongs to save(A), and can be reached from one of the states of W, where W is the state set containing all next states of state A.

For the example shown in Figure 5, the *Save-Subgraph* of state A, without output labels, is shown in Figure 6.

We now present an algorithm to obtain what we call the *Corresponding Subgraph* of the *Save-Subgraph* of a state A. The *Corresponding Subgraph* will then be substituted for the appropriate SAVE construct in the *SDL-graph*. When this procedure is carried out for every state containing a SAVE we obtain an equivalent *SDL-graph* which does not contain any SAVE.

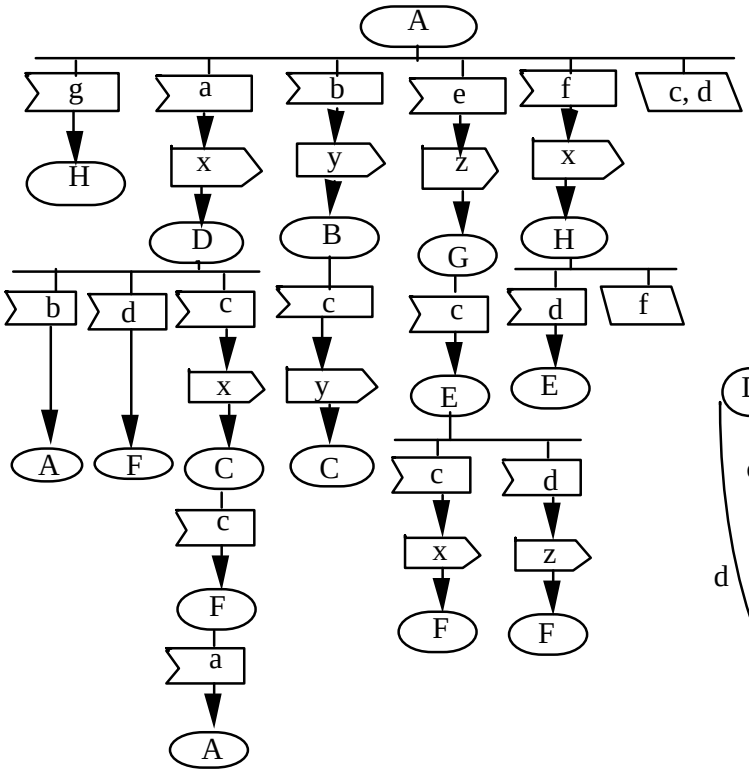

Figure 5. An example of SDL-machine

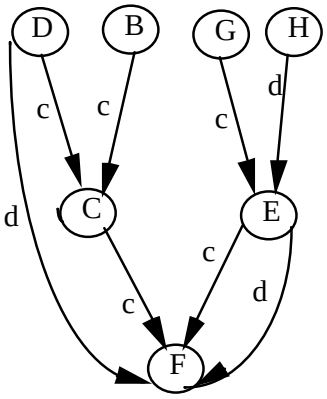

Figure 6. Save-subgraph

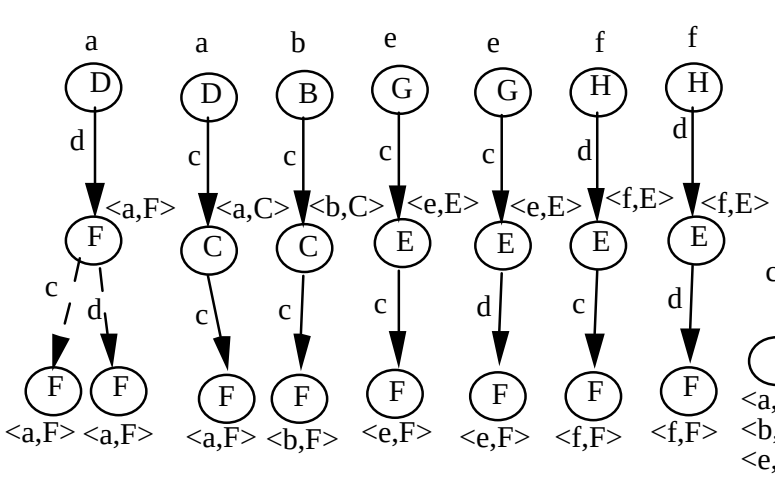

Figure 7. Splitting, extending and marking

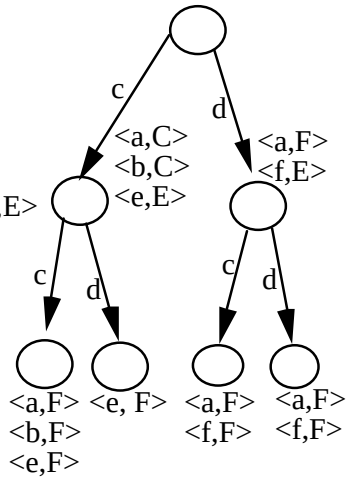

Figure 8. Merging

The algorithm given below terminates after a finite number of steps and determines the *Corresponding Subgraph* of a *Save-Subgraph* if the following conditions are satisfied for the given state A (conditions for constructing the *Corresponding Subgraph*):
Condition (I). There is no directed cycle in the *Save-Subgraph* of "A";
Condition (II). For each state S of the *Save-Subgraph*, the intersection between save(S) and save(A) is empty.

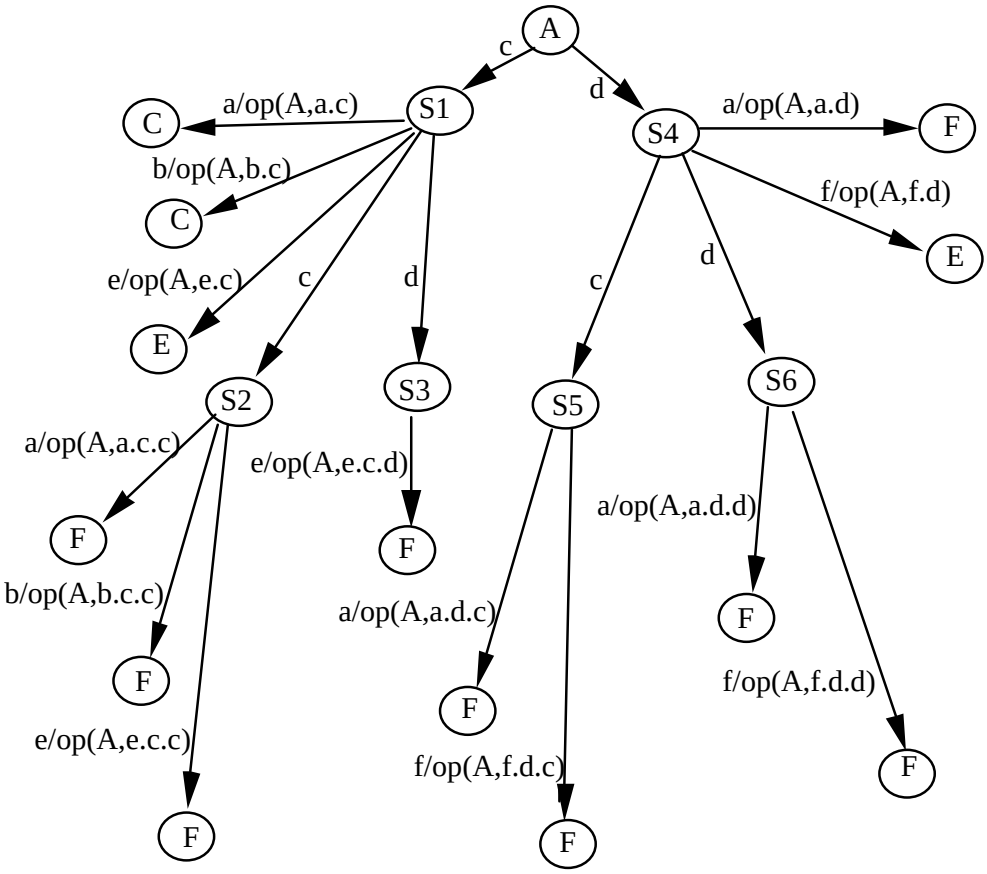

Figure 9. Naming and Adding

ALGORITHM 1: Construction of the *Corresponding Subgraph* of the *Save-Subgraph*.
Input: *Save-Subgraph* of a state A.
Output: *Corresponding Subgraph* of the *Save-Subgraph* of A.
Conditions of applicability: *Save-Subgraph* of A satisfies Conditions(I) and (II) above.

Step 1 (splitting): Enumerate all maximal directed paths of the *Save-Subgraph* to get a forest with each chain corresponding to a maximal directed path; the nodes labels and input labels of the edges are left unchanged. (See the plain lines part of Figure 7, which are chains.)

Step 2 (extending): Let L be the length of the longest chain. Extend all chains whose length is shorter than L to trees in the following way. A tree (initially a chain) is extended by adding |save(A)| directed edges with different input labels of save(A) to the leaves of the tree(note: |save(A)| denotes the cardinality of the set save(A).); this extension sub-step is applied repeatedly until the length of the tree is equal to L. (See Figure 7, the dashed edges are the extended edges)

Step 3 (marking): First, mark each root of the forest with the input label leading to the root from A through a transition. Then, mark each node which is not a root with a ordered pair <i, S> in which i is the mark of its root and S is the state name of the node. (See Figure 7).

Step 4 (merging): First, merge all roots of the forest to obtain a single root. Second, in a top-down fashion starting from the root, for every node, if the node has several outgoing edges with the same input label, say "a", merge all such edges and their corresponding end nodes into one edge and one end node; the edge thus obtained is given the label "a" and the mark of its end node is the set of the different marks (pairs) of the merged nodes.(Note that the end node thus obtained does not contain any duplicate pairs) Continue the second phase until no further progress can be made. (See Figure 8).

Step 5 (naming and adding): First, assign name A to the root of the merged tree. Then assign distinct names to the other nodes in the merged tree -- these names should be different from those in the set K. Next, for every node P and for every pair <i, S> of the node P, add a directed edge leading from the node P to state S with the input label being i and output label being op(A, i.Iseq) where the Iseq is the input label sequence leading from the root of the tree to the node P. (See Figure 9)

THEOREM 1: If the *Save-Subgraph* of state A satisfies Conditions (I) and (II), then its *Corresponding Subgraph* is a deterministic FSM.
Proof: See Appendix.

ALGORITHM 2: *SDL-machine* transformation for one state with SAVE.
Input: *SDL-machine* and state A.
Output: *SDL-machine*.
Conditions of applicability: *Save-Subgraph* of A satisfies conditions(I) and (II).

For the state A in an *SDL-machine*:
1) Find its *Save-Subgraph*;
2) Find its *Corresponding Subgraph*;
3) Replace the save(A) by the empty set and add the *Corresponding Subgraph*.

For the example shown in Figure 5, using Algorithm 2, we obtain the equivalent *SDL-machine* shown in Figure 10. The FSM of the lower part in Figure 10 is equivalent to the FSM in Figure 9.

THEOREM 2: Given an *SDL-machine* M and a state A, the input/output behavior of the *SDL-machine* obtained using Algorithm 2 is equivalent to M if Conditions (I) and (II) are satisfied for state A.
Proof: See Appendix.

If each state of the *SDL-machine* satisfies Conditions (I) and (II) then the following algorithm produces an equivalent FSM.

ALGORITHM 3: *SDL-machine* transformation.
Input: *SDL-machine* with SAVE's.
Output: *SDL-machine*.

1) Check each state of the *SDL-machine* to find a state which satisfies Conditions (I) and (II), then eliminate it by using Algorithm 2.
2) Continue this procedure until no such state is found.

It is easy to see from Theorem 2 that if an *SDL-machine* can be transformed to a FSM by Algorithm 3, then they have the same input/output property. If Algorithm 3 terminates and the resulting *SDL-machine* still contains SAVEs, we have to use the approach presented in Section 3.3.

The following result shows that Conditions (I) and (II) are both necessary and sufficient if the SAVE constructs associated with each state can have at most one input signal.

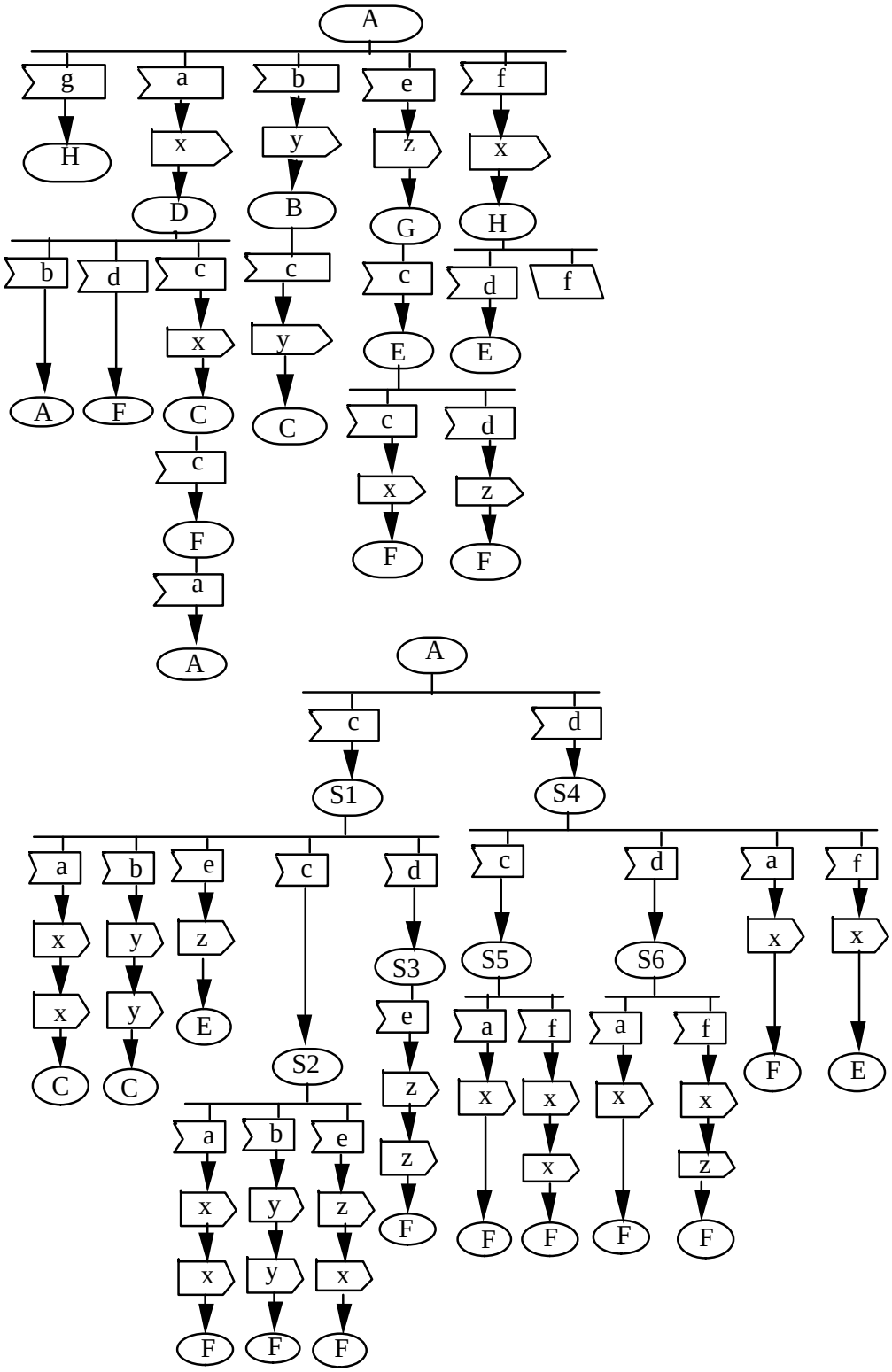

Figure 10. An Equivalent FSM

THEOREM 3: If each state A of the *SDL-machine* has |save(A)|<=1 and there is an equivalent FSM, then Conditions (I) and (II) are satisfied at least for one state, and Algorithm 3 will terminate with an equivalent FSM.
Proof: See Appendix.

3.3. Approximately Equivalent Transformation

For those *SDL-machine*s which are not equivalent to FSMs, we try to transform them into FSMs by cutting the cycles of the *Save-Subgraph*s in order to adopt the testing methods for finite state machines. For this purpose, we define in the following a transformation procedure using three algorithms which are similiar to those of Section 3.2. In the first algorithm below, we assume that the *Save-Subgraph* of a given state A may contain cycles. The algorithm cuts the cycles in the subgraph to obtain directed paths in its first step. The remaining steps are the same as in Algorithm 1. We call the resulting subgraph the *Corresponding Subgraph*-B of the *Save-Subgraph* of A.

ALGORITHM 1B: Construction of the *Corresponding Subgraph*-B of *Save-Subgraph* by cutting cycles.
Input: *Save-Subgraph* and a state A.
Output: *Corresponding Subgraph*-B.

Step 1 (splitting): Let U be the length of a longest elementary path in the *Save-Subgraph*. Enumerate all maximal directed paths of the *Save-Subgraph* with their lengths less than or equal to U to get a forest with each chain corresponding to a path, and keeping the labels of nodes and input labels of edges unchanged. (See Figures 11, 12, 13, where the U=2.) The U should be selected to make every edge of the *Save-Subgraph* be one of the edges of the forest.

The remaining four steps of this algorithm are the same as steps 2,3,4,and 5 of Algorithm 1.

The above splitting step implies cutting the cycles of the *Save-Subgraph*. The following algorithms, Algorithms 2B and 3B are similar to Algorithms 2 and 3 respectively.

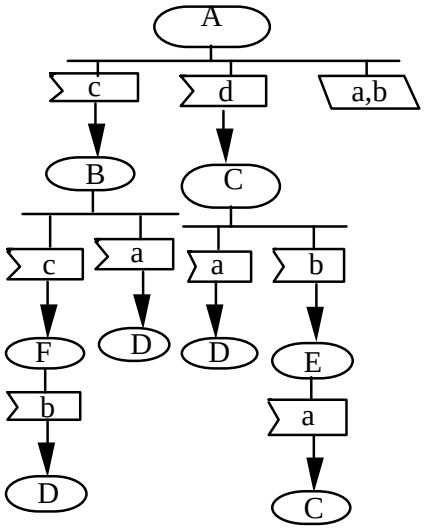

Figure 11. An example of an SDL-machine

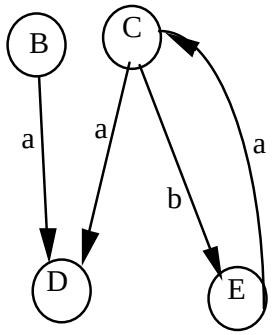

Figure 12. Save-subgraph of State A

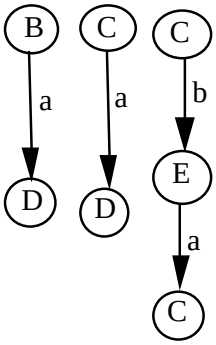

Figure 13. Splitting when U=2

ALGORITHM 2B: *SDL-machine* transformation for one state with SAVE by cutting paths.
Input: *SDL-machine* and and a state A.
Output:*SDL-machine*.

For a state A in an *SDL-machine*:
1) Find its *Save-Subgraph*;
2) Find its *Corresponding Subgraph*-B;
3) Replace save(A) by the empty set and add the *Corresponding Subgraph*-B.

ALGORITHM 3B: *SDL-machine* transformation to an approximately equivalent FSM.
Input: *SDL-machine* with SAVE's.
Output: *SDL-machine* without SAVE.

1) First, apply Algorithm 3 ;
2) Check each state of the *SDL-machine* to find a state A with save(A) <> empty, then eliminate it using Algorithm 2B. Continue this procedure until no such state is found.

4. TEST DESIGN

The FSM of SDL has an infinite input queue, but its implementation can only have a finite input queue. Therefore if there is no flow control for the input queue, the implementation is incorrect according to its SDL specification. The fault coverage should be analyzed based on both the test selection methods for FSM's and the flow control for the input queue. The flow control criteria should be given in the specification. For ease of representation, we assume the following definitions.

DEFINITION: A *prime input sequence* (*P-sequence*) of a given input signal sequence at

state A. For an input signal sequence a1.a2.....an, its *P-sequence* at the state A is an input signal sequence which is obtained by eliminating all input signals in a1.a2.....an which can only trigger implied transitions when the sequence is applied to state A. An input sequence a1.a2.....an is a *P-sequence* at state A if no implied transition occur when a1.a2.....an is applied at state A.

For the example shown in Figure 5, let the *SDL-machine* be in state A, (1) the *P-sequence* of a.a.b.b.b is a.b.b, (2) the *P-sequence* of a.a**i.b.b**j(i,j are any two of natural numbers with j>o) is a.b.b, (4) f**i is a *P-sequence* at state A, and (4) the *P-sequence* of f**i.d is f.d.

DEFINITION (*minimum capacity of queue*): Let Iseq.a be a longest input sequence satisfying the following conditions: There exists a state A such that
(1) "Iseq" belongs to (save(A))*,
(2) "a" belongs to out(A),
(3) Iseq.a is *P-sequence*.

We say that the length of Iseq.a is the *minimum capacity*. It can be either a finite integer or infinite.

We now present a method for test selection using our transformation approach. Suppose that the input queue would not have more than U (U is the upper boundary of the queue) input signals according to the flow control criteria given in the specification; the test suite of an *SDL-machine* can be generated as follows.

Step 1: Transform the *SDL-machine* to a FSM using Algorithm 3 (direct transformation) if it is possible; otherwise apply Algorithm 3B (Transformation by cutting cycles).

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 [4], the WP-method [7] and or those given in [5,6].

Step 3: Generate a boundary test case which is an input sequence of the form a**(U-1).b (a sequence of U-1 "a"'s followed by "b") such that "a" belongs to save(A) for some state A, and "b" belongs to out(A).

Consider the example shown in Figure 5. Suppose the upper boundary is 3. Then c.c.a starting from state A is the boundary test case. If the length of the queue is less than 3, the "a" will be lost after inputting c.c.a and the input flow control fault can be detected by observing the corresponding erroneous output.

The fault coverage of the test suite is the following:

(a) If the upper boundary U is greater than or equal to the *minimum capacity* and the *SDL-machine* can be transformed into an FSM by Algorithm 3 (direct transformation), then the above test suite can detect any output fault, any transfer fault, any save input fault, and any multiple saved inputs fault (up to U).

(b) Otherwise if the upper boundary U is greater than or equal to 2 and the *SDL-machine* is transformed into an FSM by Algorithm 3B (transformation by cutting cycles), then the above test suite can detect the above faults, except for multiple saved inputs faults and transfer faults.

In the second case, test case selection is based on the transformed FSM which is an approximation of the original *SDL-machine*. Therefore there is no guarantee for complete fault coverage.

5. CONCLUSION

This paper proposes a fault model for SDL specifications containing SAVE constructs, and presents a method for generating a test suite based on the fault model. The most important step of our method is to transform the *SDL-machine* to an equivalent FSM which preserves the relation between input and output sequences. For the case that each SAVE contains only a single input signal, we can transform every *SDL-machine* to a FSM if such an FSM exists. But for the case of SAVE containing several input signals, our algorithm does not work for all *SDL-machine*s for which an equivalent FSM exists. The method could be implemented by a software tool.

Acknowledgements: The authors would like to thank many people, in particular Prof. Sarikaya, Mr. Cheng Wu, Mr. Mingyu Yao and Mr. Martin Dubuc for helpful discussions. This work was supported by the IDACOM-NSERC-CWARC Industrial Research Chair on Communication Protocols at the University of Montreal (Canada).

1 R.Saracco, Course on SDL, CSELT, 1987.
2 Gang LUO & Junliang CHEN, " Investigation & Testing for SDL SAVE Function" , Journal of Beijing Univ. of Posts & Telecommunications, Vol.12, No.4, December, 1989.
3 Anne Bourguet-Rouger & Pierre Combes, "Exhaustive Validation and Test Generation in Elivis", SDL Forum'89.
4 T.S.Chow, "Testing Software Design Modeled by Finite-State Machines, IEEE Trans. on Software Eng., Vol. SE-4, No.3, 1978.
5 K.Sabnani & A.T.Dahbura, "A New Technique for Generating Protocol Tests", ACM Computer Communication Review, Vol.15, No.4, 1985, pp.36-43.
6 B.Sarikaya & G.v.Bochmann, "Some Experiences with Test Sequence Generation for Protocols", Protocol Specifications, Testing, and Verification II, 1982.
7 S.Fujiwara, G.v. Bochmann,F.Khendek,M.Amalou & A.Ghedamsi, "Test Selection Based on Finite State Models", accepted by IEEE Trans. On Software Engineering.
8 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.
9 F. Belina and D. Hogrefe, "The CCITT-Specification and Description Language SDL", Computer Networks and ISDN Systems, Vol. 16, pp.311-341, 1989.

APPENDIX

THEOREM 1: If the *Save-Subgraph* of state A satisfies Conditions (I) and (II), then its *Corresponding Subgraph* is a deterministic FSM.

Outline of proof:

1) It is easy to see that *Corresponding Subgraph* is a tree, and the sub-tree consisting of its root, its internal nodes and the edges between the nodes corresponds to a deterministic FSM.

2) After the merging step and eliminating duplicate pair in every node mark, for any node mark there does not exist two pairs such that the first parts(input signals) are equal and the second parts (states) are different from each other. Assume the contrary. Then there exist two pairs <i,s1>, <i,s2> such that s1 <> s2. This implies that there are two directed paths from same starting state to different object states with the same input sequence; hence the *Save-*

Subgraph is nondeterministic; this is contrary to the assumption that the original FSM is deterministic.

THEOREM 2: Given an *SDL-machine* M and a state A, the input/output behavior of the *SDL-machine* obtained using Algorithm 2 is equivalent to M if the Conditions (I) and (II) are satisfied for state A.

Outline of proof:

According to the transformation, we only need to prove that for all Iseq.a with Iseq belonging to (save(A))* and "a" belonging to out(A), there is

<A>@[Iseq.a] of F1 = <A>@[Iseq.a] of F2 and
op(A,Iseq.a) of F1 = op(A,Iseq.a) of F2.

First part: We need to show that

if <A>@[Iseq.a] of F1 = <S> and op(A,Iseq.a) of F1 = Oseq
then <A>@[Iseq.a] of F2 = <S> and op(A,Iseq.a) of F2 = Oseq

Let Iseq1.a be the *P-sequence* of Iseq.a in state A of F1, then we have
<A>@[Iseq.a] of F1 = <A>@[Iseq1.a] of F1 = <A>@[a.Iseq1] of F1 and
op(A,Iseq.a) of F1 = op(A,Iseq1.a) of F1 = op(A,a.Iseq1) of F1 ..... (1)

From the Conditions (I) and (II), there is an elementary path(without cycles) with its input label being Iseq1 in the *Save-Subgraph*. Furthermore, there is a path with its input label being Iseq1.a in the *Corresponding Subgraph* and

<A>@[Iseq1.a] of F2 = <A>@[a.Iseq1] of F1
and op(A,Iseq1.a) of F2 = op(A,a.Iseq1) of F1 ..... (2)

From the *Corresponding Subgraph*, we also have

<A>@[Iseq1.a] of F2 = <A>@[Iseq.a] of F2
and op(A,Iseq1.a) of F2 = op(A,Iseq.a) of F2 ..... (3)

From (1),(2),(3), we have

<A>@[Iseq.a] of F1 = <A>@[Iseq.a] of F2
and op(A,Iseq.a) of F1 = op(A,Iseq.a) of F2

Second part: We need to show that

if <A>@[Iseq.a] of F2 = <S> and op(A,Iseq.a) of F2 = Oseq
then <A>@[Iseq.a] of F1 = <S> and op(A,Iseq.a) = Oseq.

The proof of the second part is similar to the first part.

THEOREM 3: If each state A of the *SDL-machine* has |save(A)|<=1 and there is an equivalent FSM, then Conditions (I) and (II) are satisfied at least for one state, and Algorithm 3 will terminate with an equivalent FSM.

Outline of Proof: If there is an *SDL-machine* having |save(A)|<=1 which can not be transformed into an equivalent FSM by Algorithm 3, we can find a *P-sequence* of infinite length for some state A of the *SDL-machine* where after the *P-sequence* is received it will be kept in the queue and cannot trigger any transition before another input signal is input.