# Software Testing Based on SDL Specifications with Save

Gang Luo, *Member, IEEE*, Anindya Das, and Gregor v. Bochmann, *Senior Member, IEEE*

***Abstract*—The signal save construct is one of the features distinguishing SDL from traditional high-level specification and programming languages. However, this feature increases the difficulties of testing SDL-specified software. We present a testing approach consisting of the following three phases: SDL specifications are first abstracted into finite state machines with save constructs, called *SDL-machines*; the resulting SDL-machines are then transformed into equivalent finite state machines without save constructs if this is possible; and, finally, test cases are selected from the resulting finite state machines. Since there are many existing methods for the first and third phases, we mainly concentrate in this paper upon the second phase and come up with a method of transforming SDL-machines into equivalent finite state machines, which preserve the same input/output relationship as in the original SDL-machines. The transformation method is useful not only for testing but also for verifying SDL-specified software.**

***Index Terms*—CCITT SDL, communication software, finite state machines, protocol conformance testing, protocol verification, software testing, and SDL-machines.**

## I. Introduction

At PRESENT, the three formal specification languages that 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 that is most widely used in industrial applications [15], [1], [20]. 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 are based on an extended finite state machine (EFSM) model. 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. On the other hand, as pointed out in [1], the save construct was the first of 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 and verifying SDL-specified software. For this reason, several SDL-based test generation methods either prohibit the use of save constructs [18], [2] or do not address them at all [17].

In the area of testing EFSM-based software, it is a common practice to first transform EFSM's into finite state machines (FSM's) by neglecting or unfolding parameters [19], [3]; testing is then conducted based on the resulting FSM's, since many effective test generation methods are available for FSM's [9]–[11]. However, since SDL specifications are based on an EFSM model but extended with save constructs, they are usually transformed into FSM's with additional save constructs [3], [4], called *SDL-machines* [5], instead of pure FSM's. The test generation methods for FSM's are not applicable to SDL-machines. Therefore methods are needed for testing SDL-machines.

Some initial efforts have been made on generating tests for SDL-machines [3]–[5]. They all use a common key idea of transforming SDL-machines into equivalent FSM's, which preserve the same input/output relationship as in the original SDL-machines. Then test cases can be generated from the resulting FSM's using existing methods. A formal method for such an equivalent transformation was presented in [4] and a similar framework was introduced informally through examples in [3]; but they cannot provide equivalent transformation for the case where a save construct has several inputs, a case which is quite common. The equivalent transformation method presented in [5] allows the existence of several inputs in a save construct. However, it is only applicable to a still-limited subset of those SDL-machines for which the equivalent FSM's exist.

In this paper we generalize the approach introduced in [5] to obtain an equivalent transformation method that works for a larger subset of SDL-machines than the one given in [5]. We first prove that not all SDL-machines can be modeled by equivalent FSM's, though we find that in our experience most SDL-machines obtained from practical SDL specifications can be modeled by equivalent FSM's. We then come out with an equivalent transformation method that works for a larger subset of SDL-machines than the one in [5]. We finally show that, for SDL-machines where every explicit transition has at least one output, our method works precisely when there is an equivalent FSM.

We generate test cases for SDL specifications in the following three phases. First, use the approaches as given in [3] and [19] to obtain SDL-machines from SDL specifications by neglecting or unfolding parameters. Second, transform the SDL-machines into equivalent FSM's using our algorithm.

Manuscript received April 1992; revised October 1993. This work was supported by the IDACOM–NSERC–CWARC Industrial Research Chair on Communication Protocols at the Université de Montréal, Montréal, P.Q., Canada. Recommended by S. H. Zweben.

G. Luo was with the Departement d'IRO, Université de Montréal, Montréal, P.Q., Canada. He is now with the Department of Computer Science, University of Ottawa, Ottawa, Ont. K1N 6N5, Canada.

A. Das and G. v. Bochmann are with the Departement d'IRO, Université de Montréal, Montréal, P.Q. H3C 3J7, Canada.

IEEE Log Number 9214488.

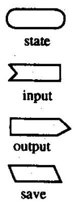


Fig. 1. SDL graphic symbols that are used in SDL-machines.

Finally, generate the test cases for the resulting FSM's by applying existing test generation methods for FSM's.

Our equivalent transformation method is also important for verifying SDL specifications. For example, SDL specifications usually need to be abstracted into FSM's for verifying a so-called *deadlock-free property*. This can be done by first abstracting SDL specifications into SDL-machines and then applying this transformation method to obtain equivalent FSM's.

The rest of the paper is organized as follows. Section II is devoted to an introduction of SDL-machines and related notations. Section III studies the equivalent transformation from SDL-machines into FSM's. Section IV handles test case selection for SDL-machines using the results of Section III and analyzes the test coverage thus obtained.

## II. Preliminaries

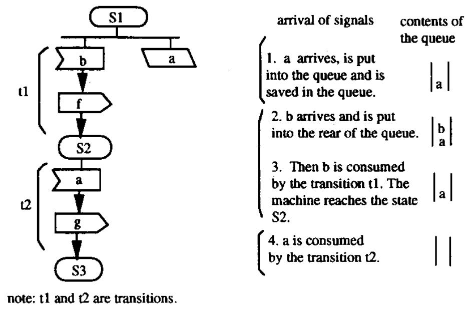
### A. Informal Description of SDL-Machines

We give in this section an informal introduction to a class of simplified SDL processes [6], [7], [1], which we call *SDL-machines* [5]. An SDL-machine is a simplified SDL process that has only the following constructs: a) *states,* b) *inputs,* c) *outputs,* d) *saves,* e) *transitions,* and f) *an input queue.* It is actually a finite state machine with the extension of an input queue and save constructs. Fig. 1 lists a subset of SDL graphic symbols that are used to present SDL-machines.

We now describe SDL-machines informally. The formal definition is presented in Section II.B.

We first describe the syntax of SDL-machines, which is given in a graphical form. An SDL-machine consists of 1) a finite number of states, each of which may have a save construct, 2) a finite number of *(explicit)* transitions, each of which has one input and zero or more outputs, and 3) an input queue. A save construct may have one or more inputs. For an SDL-machine, the input of every transition of a given state is different from the inputs of any other transitions of the same state, and it is not any input specified in the save construct of the same state. Thus SDL-machines are deterministic state machines. This means that, given an SDL-machine, for any state $S$, and for any input sequence $x$, the machine always produces exactly the same output sequence each time $x$ is applied to $S$. Fig. 2(a) shows an example of an SDL-machine.

We now describe the behavior of SDL-machines. Given an SDL-machine, every arriving input is first placed into the rear of the input queue. Assume that the queue is not empty and the machine is in a state $S$; then the following cases may arise:



S1
b
a
t1
f
S2
a
t2
g
S3
note: t1 and t2 are transitions.
arrival of signals
contents of the queue
1. a arrives, is put into the queue and is saved in the queue.
a
2. b arrives and is put into the rear of the queue.
b
a
3. Then b is consumed by the transition t1. The machine reaches the state S2.
a
4. a is consumed by the transition t2.
(a)
(b)

Fig. 2. Illustration of an SDL-machine. (a) An SDL-machine. (b) Illustration of behavior of the SDL-machine.

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

**Case 2:** Among all inputs in the queue that are not specified in the save construct of the state $S$, there is an input $b$ that is the nearest to the front of the queue. In this case, the following two situations arise: a) If $b$ is attached to one of outgoing (explicit) transitions from $S$, it will be removed from the queue; the corresponding transition will be performed ($b$ is said to *be consumed* by the transition), and the SDL-machine will move to a next state. b) If $b$ is not attached to any outgoing (explicit) transition from $S$, it will be removed from the queue, but no (explicit) transition will be performed. In this situation, the input $b$ is said to be consumed by an *implied transition* that starts from and goes back to the same state $S$ without any output being sent.

If a given SDL-machine does not have any save construct, the input queue becomes an first-in-first-out (FIFO) queue. The save constructs make the queue non-FIFO.

We illustrate the functioning of SDL-machines with the example shown in Fig. 2. Assume that the machine is initially in the state $S1$ with the queue empty. An input $a$ arrives; it is kept in the queue because $a$ appears in the save construct of $S1$. Then an input $b$ arrives; it is consumed by the transition $t1$, leading the machine from the state $S1$ to the state $S2$ with the output $f$ sent. Finally, the $a$ in the queue is consumed by the transition $t2$, leading the machine to the state $S3$ with the output $g$ sent.

### B. Formal Definitions of SDL-Machines and Related Concepts

We formally define in this section the syntax and behavior of SDL-machines, as well as concepts and notations related to SDL-machines. The syntax of SDL-machines is presented in a symbolic form, as follows.

*Definition:* SDL-Machines.

An *SDL-machine* is a 6-tuple ($K, I, O,$ *saveset,* $T, S_0$) with an input queue where we have the following.

1) $K$ is a finite set of symbols, called *states.*

2) $I$ is a finite set of symbols, called *inputs.*

3) $O$ is a finite set of symbols, called *outputs*.
4) *saveset* is a function, *saveset*: $K \longrightarrow$ powerset($I$).
5) $T$ is a function, called *transition function*. $T : \mathbb{D} \longrightarrow K \times O^*$ where $\mathbb{D} = \{< S, a > \in K \times I | a \notin saveset(S)\}$ and $O^*$ denotes the set of sequences over $O$.
   We say that the SDL-machine has a *transition* from $S_1$ to $S_2$ with input $a$ and the output sequence $w$, if $T(S_1, a) = < S_2, w >$. This is also denoted by $S_1 - a/w \rightarrow S_2$. If $w = \lambda$ ($\lambda$ stands for the empty sequence) and $S_1 = S_2$ (same state), we call the transition an *implicit transition* (Note: In SDL specifications, such transitions are not explicitly defined); the other transitions are called *explicit transitions*.
6) $S_0 \in K$ is the *initial state* of the SDL-machine. □

From this definition, a transition has only a single input but may have an output sequence, as assumed in SDL. States, inputs, outputs, transitions, and savesets in the symbolic notations correspond to states, inputs, outputs, transitions, and saves in the graphical notations of SDL-machines. As an example, for the SDL-machine of Fig. 2 we have

$K = \{S1, S2, S3\}, I = \{a, b\}, O = \{f, g\}, saveset(S1) = \{a\}, saveset(S2) = saveset(S3) = \emptyset$ ($\emptyset$ stands for an empty set), the initial state $S1$, explicit transitions: $S1 - b/f \rightarrow S2$ and $S2 - a/g \rightarrow S3$, and implied transitions:$S2 - b/\lambda \rightarrow S2, S3 - a/\lambda \rightarrow S3$ and $S3 - b/\lambda \rightarrow S3$.

To make definitions less cumbersome, we assume that all SDL-machines being discussed are denoted as $(K, I, O, saveset, T, S_0)$ unless specified explicitly. We use the notation "." to represent the concatenation of two input sequences or two output sequences.

Given a pair $[S, x] \in K \times I^*$, we call $[S, x]$ a *global state*, which represents the fact that the SDL-machine is in the state $S$ with the input sequence $x$ in the input queue. $K \times I^*$ is the set of all possible global states. Furthermore, we say that a global state $[S, x]$ is *stable* if $x \in saveset(S)^*$; and $\mathbb{G}$ denotes the set of all stable global states for an SDL-machine (note: $\mathbb{G} \subseteq K \times I^*$). A global state that is not stable is said to be *unstable* . If an SDL-machine is in a stable global state, then it is waiting in this global state and cannot consume any input in its queue before an additional input is received.

We now define the behavior of SDL-machines using three functions: @, *queue* and *out*, called *transfer function, queue function* , and *output function*, respectively. $S_1@x = S_2$ and *queue* $(S_1, x) = z$ mean that, assuming an SDL-machine to be in the state $S_1$ with $\lambda$ in the input queue initially, after the input sequence $x$ is applied, the machine will eventually arrive in the state $S_2$ with $z$ in the input queue such that no inputs of $z$ can be consumed if no further inputs are received, i.e., $[S_2, z]$ is a stable global state. Furthermore, $out(S_1, x)$ stands for the output sequence eventually produced after applying the input sequence $x$ to $S_1$ when the machine is initially in the state $S_1$ with $\lambda$ in the input queue. The formal definitions are given below.

TABLE I
NOTATIONS FOR SDL-MACHINES

| Notations | Meaning |
|---|---|
| $a, b, c, d, e$ | inputs or outputs |
| $v, w, x, y, z$ | input or output sequences |
| $a^i, \quad v^i$ | $i$ is a natural number, $a^i$ denotes a sequence of $a$ of length $i$,$v^i$ denotes a sequence of $v$ of length i. That is, $a^0 = \lambda, v^0 = \lambda$; and $a^i = a^{i-1}.a$, $v^i = v^{i-1}.v$ for $i \geq 1$. |
| $S, Q, P$ | states |
| $in(S)$ | $in(S) = \{a \vert \exists S - a/w \rightarrow S_1 (S - a/w \rightarrow S_1$ is an explicit transition). That is, $in(S)$ is the set of all inputs each of which is attached to an explicit transition from the state $S$. |
| $pref$(W) | $pref$ is a function, called *prefix function*, $pref$: powerset($I^*$) $\longrightarrow$ powerset($I^*$). Given $W \in powerset(I^*)$, $pref(W) = \{x_1 \vert \exists x \in W \exists y \in (x = x_1 . y)\}$. |
| $proj(x)$ | $proj$ is a function, called *projection function*, $proj$:$I^* \longrightarrow$ powerset($I^*$). Given $x \in I^*, proj(x) = \{y \in I^* \vert y$ is a sub-sequence of $x$, obtained by deleting zero or more inputs in $x\}$. |

*Definition:* Transfer function "@," queue function "queue," and output function "out."

Given an SDL-machine, assume that $S_1, S_2 \in K, x = a_1 \cdots a_{i-1} \cdot a_i \cdot a_{i+1} \cdots a_n \in I^* (a_i \in I, \text{for } i = 1, 2, \cdots, n)$ and $w \in O^*$. @: $K \times I^* \longrightarrow K, queue : K \times I^* \longrightarrow I^*$ and $out : K \times I^* \longrightarrow O^*$ are defined as follows:

i) if $x \notin saveset(S_1)^*$, then
   a) $S_1@a_1 \cdot a_2 \cdots a_i \cdots a_n = S_2@a_1 \cdots a_{i-1} \cdot a_{i+1} \cdots a_n$
   b) $queue(S_1, a_1 \cdot a_2 \cdots a_i \cdots a_n) = queue(S_2, a_1 \cdot a_{i-1} \cdot a_{i+1} \cdots a_n)$
   c) $out(S_1, a_1 \cdot a_2 \cdots a_i \cdots a_n) = w.out(S_2, a_1 \cdots a_{i-1} \cdot a_{i+1} \cdots a_n)$ where $a_1 \cdot a_2 \cdots a_{i-1} \in saveset(S_1)^*$, and $S_1 - a_i/w \rightarrow S_2$.

ii) if $x \in saveset(S_1)^*$, then
   a) $S_1@x = S_1$
   b) $queue(S_1, x) = x$
   c) $out(S_1, x) = \lambda$. □

$S_1@x = S_2$ and $queue(S_1, x) = z$ imply that if an SDL-machine is initially in the global state $[S_1, x]$, then the machine will eventually arrive in the stable global state $[S_2, z]$, without receiving more inputs. Given a global state $[S, x]$ and an input sequence $y$, we say that $y$ *leads* the machine from $[S, x]$ to a stable global state $[S_1, z]$ if $S@x.y = S_1$ and queue$(S, x.y) = z$.

As an example, for the SDL-machine shown in Fig. 2, we have
$S1@a.a = S1$, $queue(S_1, a.a) = a.a$, $S1@a.a.b = S3$, $queue(S_1, a.a.b) = \lambda$, $S1@b = S2$, and $queue(S_1, b) = \lambda$; $out(S1, a.a) = \lambda$, $out(S1, a.a.b) = f.g$, and $out(S1, b) = f$.

For the sake of convenience, we introduce several other notations for SDL-machines in Table I.

We note that *pref (W)* is the set of all possible prefixes of sequences in the set $W$; for $W = \{a.b, a.c, b\}$, we have *pref*$(W) = \{a.b, a, a.c, b, \lambda\}$. *proj* $(x)$ is the set of all possible sub-sequences of the sequence $x$ which is obtained by deleting zero or more inputs in $x$. For example, *proj*$(a.b.c) = \{\lambda, a, b, c, a.b, a.c, b.c, a.b.c\}$.

*Definition:* Initially connected SDL-machines.

A given SDL-machine is said to be *initially connected* if all states are reachable from the initial state $S_0$ through a sequence of transitions. □

Without loss of generality, we assume that all SDL-machines considered in the rest of the paper are initially connected. If a given machine F is not initially connected, we may consider a submachine which is a portion of F consisting of all states with their save constructs, and transitions that are reachable from the initial state of F. The unreachable portion of the machine does not affect the input/output behavior of the machine.

We note that in the case that the SDL-machine has no saveset, i.e. $\forall S \in K(saveset(S) = \emptyset)$, the machine is equivalent to a traditional FSM. In fact, in this case, the explicit and implicit transitions define a transition for each (state, input) pair. Even if the speed of the arrival of inputs is fast compared with the processing speed of the machine, and the inputs may "queue up" in the input queue, the input/output behavior, in terms of the output sequence produced for a given input sequence, is the same independently of the proceeding speed of the machine. Therefore we simply call in the following an SDL-machine without savesets an FSM. In the following we show how a general SDL-machine (with save) can be transformed into an equivalent FSM and how the testing methods developed for FSM's can be applied to general SDL-machines.

Consider an SDL-machine where the input queue contains as input sequence $a_1 \cdot a_2 \cdots a_i \cdots a_n$, and the machine is in state $S$. As mentioned in the informal description of SDL-machines given earlier, an input $a_i$ in the queue may eventually be removed and trigger a (explicit or implicit) transition, say $S_1 - a_i/w \rightarrow S_2$. In this case, we say that $a_i$ will be *consumed by the transition* $S_1 - a_i/w \rightarrow S_2$ *when* $a_1 \cdot a_2 \cdots a_i \cdots a_n$ *is applied to the state S.* We give a formal definition as follows:

*Definition:* An input consumed by an implied or explicit transition.

Given a global state $[S, x]$ with $x = a_1.a_2 \cdots a_i \cdots a_n$, let $queue(S,x) = a_{k1}.a_{k2} \cdots a_{km}$. If $i \notin \{k1, k2, \cdots, km\}$, we say that the $a_i$ of $x$ will be *consumed by a transition when* $x$ *is applied to the state S.* In this case, for the $a_i$, there must exist $S_1 - a_i/w \rightarrow S_2$, $x_1 \in saveset(S_1)^*$, and $x_2 \in I^*$ such that $S@a_1 \cdot a_2 \cdots a_i \cdots a_n = S_1@x_1 \cdot a_i \cdot x_2 = S_2@x_1 \cdot x_2$ (note: i)$S_1@x_1 \cdot a_i \cdot x_2$ is an intermediate step for deriving $S@a_1 \cdot a_2 \cdots a_i \cdots a_n$, ii). $S_1, S_2, x_1$, and $x_2$ are unique). If the $S_1 - a_i/w \rightarrow S_2$ is an explicit transition (i.e., $a_i \in in(S_1)$), then we say that the input $a_i$ of $x$ will be *consumed by an explicit transition when* $x$ *is applied to a state S*; and if $S_1 - a_i/w \rightarrow S_2$ is an implied transition (i.e., $a_i \notin in(S_1)$), we say that the input $a_i$ will be *consumed by an implied transition.*

For the example of the SDL-machine shown in Fig. 2, let $a_1.a_2.a_3 = a.a.b$; $a_2$ will be consumed by an implied transition (i.e., by $S3 - a/\lambda \rightarrow S3$ ) when $a_1.a_2.a_3$ is applied to $S1.a_1$, and $a_3$ will be consumed by explicit transitions (i.e., by $S2 - a/g \rightarrow S3$ and $S1 - b/f \rightarrow S2$, respectively) when $a_1.a_2.a_3$ is applied to $S1$. For $a_1.a_2 = a.a$, none of the inputs of $a_1.a_2$ will be consumed by any transition when $a_1.a_2$ is applied to $S1$.

## C. The Equivalence Relation for SDL-Machines

We present in this section a conformance relation for SDL-machines. Before generating test cases, one should answer the following question: What is the conformance relation to be checked between a specification and its corresponding implementation? Under a black-box-testing strategy where only the inputs and outputs of implementations are accessible, we answer this question by defining a so-called *equivalence* relation, which requires that *two SDL-machines* (a specification and its implementation) *produce the same output sequence for every input sequence.* This relation is the same equivalence relation for finite state machines [9], [10] and is formally presented as follows:

*Definition:* Equivalence between global states of SDL-machines.

Given two SDL-machines F1 and F2 that have the same input set $I$ and the same output set $O$, let $out_1$ and $out_2$ be the output functions of F1 and F2; for two global states $[S_i, x]$ and $[S_j, y]$ in F1 and F2, respectively,

$[S_i, x]$ and $[S_j, y]$ are said to be *equivalent* if $\forall z \in I^*(out_1(S_i, x.z) = out_2(S_j, y.z))$. □

In this definition, F1 and F2 can be the same machine. Therefore, for two global states $[S_i, x]$ and $[S_j, y]$ in F1, $[S_i, x]$ and $[S_j, y]$ are equivalent if $z \in I^*(out_1(S_i, x.z) = out_1(S_j, y.z)$ ).

*Definition:* Equivalence between SDL-machines.

Given two SDL-machines F1 and F2 that have the same input set $I$ and output set $O$, assuming that $S_{01}$ and $S_{02}$ are the initial states in F1 and F2, respectively,

F1 and F2 are said to be *equivalent* if their initial global states $[S_{01}, \lambda]$ and $[S_{02}, \lambda]$ are equivalent.

According to this definition, two SDL-machines F1 and F2 are equivalent if and only if $\forall x \in I^*(out_1(S_{01}, x) = out_2(S_{02}, x))$, where $out_1$ and $out_2$ denote the output functions of F1 and F2. The definition of equivalence relation serves as a guide to the development of test case generation methods.

## III. Transforming SDL-Machines into Equivalent FSM's

On the basis of the equivalence definition given before, we study in this section a method of transforming a given SDL-machine into an equivalent FSM. The equivalent FSM is obtained by deleting all save constructs, by introducing additional states that do not have any save construct, and by incorporating additional transitions.

### A. An Example of an SDL-Machine Without an Equivalent FSM

We show in the following that not all SDL-machines can be modeled by equivalent FSM's, though equivalent FSM's can be found for SDL-machines resulting from most practical applications. An example of an SDL-machine for which there does not exist any equivalent FSM is given in Fig. 3. The following arguments show that this SDL-machine does not have any equivalent FSM.

Consider the SDL-machine shown in Fig. 3. We have $out(S1, a^i.b) = x.y^i$, for $i = 1, 2, \cdots$; i.e., one of

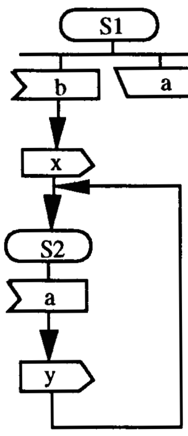


Fig. 3. An example of an SDL-machine without any equivalent FSM.

the different output sequences $x.y, x.y^2, \cdots, x.y^i, \cdots$ is produced after input $b$ is applied to the stable global states $[S1, a], [S1, a^2], \cdots, [S1, a^i], \cdots$, respectively. Therefore, from the definition of the equivalence of global states, none of the stable global states $[S1, a], [S1, a^2], \cdots, [S1, a^i], \cdots$ are equivalent. The input sequences $a, a.a, \cdots, a^i, \cdots$, lead this machine from the initial global state $[S1, \lambda]$ to $[S1, a], [S1, a^2], \cdots, [S1, a^i], \cdots$, respectively. Thus, for any SDL-machine equivalent to the one shown in Fig. 3, $a, a.a, \cdots, a^i, \cdots$ must lead the machine from its initial global state to a set of stable global states $T_1, T_2, \cdots, T_i, \cdots$ that are equivalent to $[S1, a], [S1, a^2], \cdots, [S1, a^i], \cdots$, respectively. None of $T_1, T_2, \cdots, T_i, \cdots$ are equivalent. Such an equivalent machine must have an infinite number of stable global states, because $i$ can take any positive integer value. An FSM has only a finite number of stable global states, however, because it does not have any save construct. (See the definition of FSM's given in Section II-B.) Therefore, the machine shown in Fig. 3 cannot be modeled by any FSM, and we obtain the following result.

*Theorem 1:* There exist SDL-machines that cannot be modeled by equivalent FSM's.

Therefore, we need to identify the classes of SDL-machines that can be modeled by equivalent FSM's and develop the equivalent transformation algorithm accordingly.

## *B. An Equivalent Transformation Method*

We first give an intuitive description of our method and then formally present it in the following three subsections. We need the following concepts for describing our method.

*Definition:* Neutral-inputs ($n$-inputs).

For a given state $S \in K$ and an input sequence $x = a_1 \cdot a_2 \cdots a_i \cdots a_k \in saveset(S)^*$, we say that the input $a_i$ of $x$ is *an n-input of x at S* if $\forall z \in I^*$ (the input $a_i$ will not be consumed by any explicit transition when $x.z$ is applied to $S$). □

This concept is based on the following intuitive idea: Given a state $S \in K$ and an input sequence $x \in saveset(S)^*$, for every $y \in I^*$, consider the application of $x.y$ to $S$. Since any $n$-input of $x$ at $S$ when consumed is consumed only by an implied transition, it does not stimulate any output and can only invoke a self-loop at a state. For the example shown in Fig. 2, let $a_1.a_2.a_3 = a.a.a$, then $a_2$ and $a_3$ are $n$-inputs of $a_1.a_2.a_3$ at the state $S1$. In the machine shown in Fig. 3, no inputs of the input sequences $a^i$ are $n$-inputs of $a^i$ at the state $S1$, for $i = 1, 2, \cdots$ .

*Definition:* Useful-subsequence ($u$-sequence).

For a given state $S \in K$ and an input sequence $x \in a_1.a_2 \cdots a_i \cdots a_k \in saveset(S)^*$, an input sequence $z$ is said to be a $u$-sequence of $x$ at $S$ if $z$ is obtained from $x$ by eliminating zero or more $n$-inputs of $x$ at $S$. (Note: $z \in proj(x)$.) □

The definition of this concept is motivated by the following intuition: Given a state $S \in K$ and an input sequence $x \in saveset(S)^*$, let $z$ be a $u$-sequence of $x$ at $S$. The same sequence of explicit transitions will be executed when $x.y$ and $z.y$ are applied to $S$, respectively. Therefore, $y \in I^*(out(S, x.y) = out(S, z.y) \& S@x.y = S@z.y)$. According to the definition of $u$-sequences, a $u$-sequence of $x$ at $S$ is not necessarily unique. For the example shown in Fig. 2, let $a_1.a_2.a_3 = a.a.a$; then $a_1, a_1.a_2, a_1.a_3$ and $a_1.a_2.a_3$ are four different $u$-sequences of $a_1.a_2.a_3$ at the state $S1$.

The concepts of $n$-inputs and $u$-sequence are not used explicitly in presenting our algorithms, but they intuitively play a key role in developing the algorithms and are used in proving the validity of the algorithms.

*Definition:* Explicitly-consumed-save-sequence ($e$-sequence).

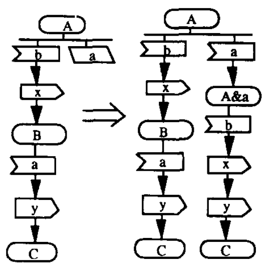
Given a state $S \in K$ and $x \in saveset(S)^*$, an input sequence $y$ is called an *e-sequence of x at S* if there exists $z \in I^*$ such that

1) all inputs in $x.z$ will be consumed when $x.z$ is applied to $S$; and
2) $y$ is derived from $x$ by eliminating all inputs in $x$ that will be consumed by implied transitions when $x.z$ is applied to $S$.

Furthermore, $x$ is said to be an *e-sequence at S* if $x$ is itself an $e$-sequence of $x$ at $S$. The set of all $e$-sequences at $S$ is denoted as $E_S$; that is, $E_S = \{x | x \text{ is an } e\text{-sequence at } S\}$. □

Intuitively, an $e$-sequence at $S$ is a sequence of inputs in $saveset(S)$ that can be consumed only by explicit transitions, and it contains no $n$-inputs at $S$. From this definition, it follows that any prefix of an $e$-sequence at a state $S$ is also an $e$-sequence at $S$; that is, $pref(E_S) \subseteq E_S$. Therefore, by the definition of $pref, pref(E_S) = E_S$. For the example shown in Fig. 5, for the states $A$ and $B, E_A = \{\lambda, a, b, a.b\}$ and $E_B = \{\lambda\}$. We note that $E_S$ is finite if and only if all $e$-sequences at $S$ are of finite length. The construction of $E_S$ is not trivial and is presented later.

We now present an intuitive description of our method. Given an SDL-machine F, if F is not an FSM, then the set of all possible stable global states $\mathbb{G}$ is infinite, representing an infinite memory. On the other hand, an FSM has only a finite number of stable global states, a finite memory. For a given stable global state $[S, x]$ in F, the $n$-inputs of the input



(a) (b)

Fig. 4. An example of illustrating equivalent transformations.

sequence $x$ at the state $S$ when consumed do not stimulate any output and only cause self-loops, since, when consumed, they are consumed only by implied transitions. Therefore, an equivalent machine does not need to remember the $n$-inputs of $x$; receiving $x$, it needs only remember one of the $u$-sequences of $x$ at the state $S$. This guides us to construct an equivalent FSM for a given SDL-machine in the following manner.

For an SDL-machine F with $K = \{S_1, S_2, \cdots, S_n\}$, let $\mathbb{G}1_i = \{[S_i, x] \| \; x \in saveset(S_i)^*\}$ be the set of all stable global states related to the state $S_i$, and $i = 1, 2, \cdots, n$ (note: $\mathbb{G} = \mathbb{G}1_1 \cup \mathbb{G}1_2 \cup \cdots \cup \mathbb{G}1_n$). To construct an equivalent FSM, we have the following two major phases:

**Phase 1:** We partition every $\mathbb{G}1_i$ into a set of classes $\mathbb{C}_{i1}, \mathbb{C}_{i2}, \cdots, \mathbb{C}_{ij}, \cdots\}$ such that all global states in the same class $\mathbb{C}_{ij}$ are equivalent. Such a $\mathbb{C}_{ij}$ is called an *equivalence class*. (Note: The partitioning is not necessarily unique.)

**Phase 2:** If every $\mathbb{G}1_i$ is partitioned into a finite number of equivalence classes $\{\mathbb{C}_{i1}, \mathbb{C}_{i2}, \cdots, \mathbb{C}_{ij}, \cdots\}$, then we construct an equivalent FSM containing states $Q_1, Q_2, \cdots, Q_m$ such that, for each $\mathbb{C}_{ij}$, there exists a state $Q_k$ such that $[Q_k, \lambda]$ is equivalent to the global states in $\mathbb{C}_{ij}$. However, if $\mathbb{G}1_i$ cannot be partitioned into a finite number of such equivalence classes, then the SDL-machine F cannot be modeled by an equivalent FSM, according to arguments similar to those for Theorem 1.

For the example shown in Fig. 4(a), the set of all stable global states related to the state $A, \mathbb{G}1_A$, can be partitioned into two equivalence classes: $\{[A, \lambda]\}$ and $\{[A, a^i]a^i \& i \geq 1\}$; each of $\mathbb{G}1_B$ and $\mathbb{G}1_C$ is partitioned into one single class of $\{[B, \lambda]\}$ and $\{[C, \lambda]\}$, respectively. $\mathbb{G}$ is partitioned into four equivalence classes: $\{[A, \lambda]\}, \{[A, a^i] | a^i \; \& \; i \geq 1\}, \{[B, \lambda]\}$ and $\{[C, \lambda]\}$, which are a finite number of classes. We construct the equivalent FSM shown in Fig. 4(b), which contains the four states $A, A\&a, B$, and $C$, such that $[A, \lambda], [A \; \& \; a, \lambda], [B, \lambda]$ and $[C, \lambda]$ in the FSM are equivalent to $\{[A, \lambda]\}, \{[A, a^i] | a^i \; \& \; i \geq 1\}, \{[B, \lambda]\}$ and $\{[C, \lambda]\}$ in the original machine, respectively.

We note that, given a state $S_i$, in order to partition each $\mathbb{G}1_i$, we need to first find $E_{Si}$, the set of all $e$-sequences at $S_i$, then partition every $\mathbb{G}1_i$ with the help of $E_{Si}$. In summary, our method consists of three major stages:

**Stage 1:** Find $E_{S_i}$ for every state $S_i$.

**Stage 2:** Construct a so-called *s-tree* (save-corresponding-tree) for every state $S_i$ with the help of $E_{S_i}$. Each tree serves as a relation for partitioning the $\mathbb{G}1_i$ into a finite set of equivalence classes.

**Stage 3:** Construct an equivalent FSM with the help of the set of equivalence classes.

The phase 1 described before is divided into the stages 1 and 2, and the phase 2 is the stage 3. The three stages are presented in detail in the next three subsections, respectively.

**Finding *e*-Sequences:**

We present in this section a method of finding $E_S$, the set of all $e$-sequences at a given state $S$ in an SDL-machine. For the convenience of presentation, we first define several concepts. In order to use the terminology of graph theory, we define a graph form of SDL-machines, called *SDL-graphs.*

*Definition:* SDL-graph.

An *SDL-graph G* is a labeled directed graph such that there exists an SDL-machine F satisfying the following:

(1) There is a one–one correspondence between the nodes in $G$ and the states in F. The node corresponding to a state $S$ in F is labeled a pair $S/saveset(S)$ that represents the state $S$ and the corresponding $saveset(S)$; and for the sake of simplicity, $S/\emptyset$ may be denoted as $S$.

(2) There is a one–one correspondence between the edges in $G$ and the explicit transitions in F. The directed edge from a node $S/saveset(S)$ to a node $Q/saveset(Q)$ corresponds to an explicit transition $S - a/x \rightarrow Q$, and is labeled the pair $a/x$, which represents the input $a$ and output sequence $x$ of the transition; $a/\lambda$ may be denoted as $a$. □

Given an SDL-machine, we can obtain a unique SDL-graph, and vice versa. States and explicit transitions in SDL-machines correspond to nodes and directed edges in their SDL-graphs. This enables us to use the terminology of graph theory for SDL-machines. Therefore, in the following, if we say edges and nodes or subgraphs of SDL-machines, we means the edges and nodes or subgraphs of corresponding SDL-graphs of the SDL-machines; similarly, the transitions and states of SDL-graphs refer to the transitions and states of the corresponding SDL-machines of the SDL-graphs.

Given a state $S$ in an SDL-machine F, in order to find $E_S$, the set of all the $e$-sequences at $S$, we construct a so-called save-affected-graph of $S$, which is a subgraph of the machine F. The save-affected-graph of $S$ intuitively captures the following notion: Let $x$ be an $e$-sequence at $S$ and any $z \in I^*$; if $x.z$ is applied to $S$, then each input of $x$, if consumed, can be consumed only by the explicit transitions in the save-affected-graph of $S$. Therefore, checking the whole machine F for the construction of $E_S$ can be reduced to checking the save-affected-graph of $S$, a portion of F.

*Definition:* The save-affected-graph of a given state.

Given an SDL-machine F, for a state $S$, an SDL-graph $G$ is said to be the *save-affected-graph* of $S$ if $G$ is the smallest subgraph of F satisfying the following:

If 1) $x$ is an $e$-sequence at $S$ and 2) $z \in I^*$ is a shortest sequence such that all inputs in $x.z$ are consumed by explicit transitions when $x.z$ is applied to $S$, then all these explicit transitions are contained in $G$. □

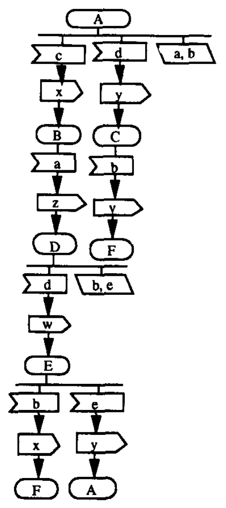


Fig. 5. An example of SDL-machine.

The *root* of the save-affected-graph of $S$ is the state $S$. In the following, we always present the save-affected-graphs in the form of SDL-graphs with all outputs and save constructs omitted.

Constructing the save-affected-graphs involves constructing what we call *save-graphs*. Given an SDL-machine F, for a state $Q$ and a set of inputs $\mathbb{Z} \subseteq saveset(Q)$, the *save-graph of $Q$ with respect to $\mathbb{Z}$* is a subgraph of F that intuitively captures the following notion: For every input sequence $x \in \mathbb{Z}^*$ and an input $a \in in(Q)$, the inputs in $x.a$, *when consumed*, can be consumed only by the transitions in the save-graph of $Q$ with respect to $\mathbb{Z}$, except that the save-graph does not contain any edge from $Q$ that has no successive edge with an input in $\mathbb{Z}$. The complete definition of save-graphs is presented constructively by an algorithm given in Appendix II.

For simplicity, we always present the save-graphs in the form of SDL-graphs with all outputs and save constructs omitted. For the example shown in Fig. 5, the save-graphs of state A with respect to $\{a, b\}$ and $\{b\}$ are shown in Figs. 6(a) and 7(b), respectively, in the form of SDL-graphs. The save-graph of state $D$ with respect to $\{b\}$ and $\{b, e\}$ is shown in Figs. 6(b) and 7(a), respectively.

We give here an intuitive description of the construction of the save-affected-graphs. Consider an SDL-machine F and a state $S$. Let all transitions in the SDL-machine F be initially unmarked. First, mark all transitions and their adjacent states in F that belong to the save-graph of $S$ with respect to $saveset(S)$. Let $G$ always represent the marked portion of F. Then, for every node $Q$ of $G$ except $S$, if $saveset(S) \cap saveset(Q) \neq \emptyset$, then construct the save-graph of $Q$ with respect to $saveset(S) \cap saveset(Q)$, say $G1$, and modify $G$ by marking all transitions and their adjacent states of $G1$. A similar procedure is repeated until no more transitions and states can be added to $G$. The resulting $G$ is the save-affected-graph of $S$. The algorithm for constructing the save-affected-graphs is given in Appendix II.

For the example shown in Fig. 5, the save-affected-graphs of states A and D are shown in Figs. 6(c) and 7(c), respectively.

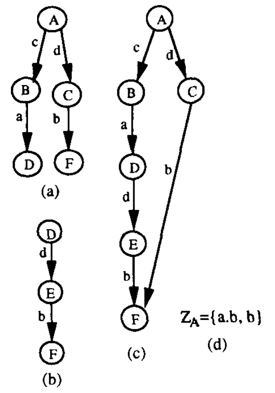


Fig. 6. (a) The save-graph of A with respect to $[a, b]$. (b) The save-graph of D with respect to $\{ b \}$. (c) The save-affected-graph of A. (d) $Z_A$ set.

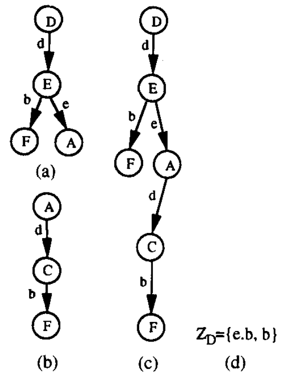


Fig. 7. (a) The save-graph of D with respect to $\{b, e\}$. (b) The save-graph of A with respect to $\{b\}$. (c) The save-affected-graph of D. (d) $Z_D$ set.

Fig. 6(c) results from Fig. 6(a) and 6(b), and Fig. 7(c) results from Fig. 7(a) and 7(b).

**Simple save-affected-graph assumption:** Given an SDL-machine and a state $S$, we say that $S$ satisfies the simple save-affected-graph assumption if

1) In the save-affected-graph of $S$, say $G$, for every state $Q$, if $Q$ has an outgoing transition in $G$ with an input in $saveset(S)$, then a) $saveset(S) \cap saveset(Q) = \emptyset$, and b) $Q$ does not have more than one incoming transition in $G$.
2) The save-affected-graph of $S$ does not have any directed cycle. □

If a given state in an SDL-machine satisfies the simple save-affected-graph assumption, we use the following approach for finding $E_S$, the set of all $e$-sequences at $S$.

**Finding all $e$-sequences at a given state:**

Given an SDL-machine, for a state $S$ that satisfies the simple save-affected-graph assumption, we construct $E_S$ as follows:

1) Construct $Z_S = \{x \in I^* | y$ is an input sequence along a maximal directed path in the save-affected-graph such that the path starts from $S$, and $x$ is derived by eliminating all inputs of $y$ that are not in $saveset(S)\}$.
2) Construct $E_S = pref(Z_S)$. □

We intuitively explain the above procedure. For a state $S$ that satisfies the simple save-affected-graph assumption, the

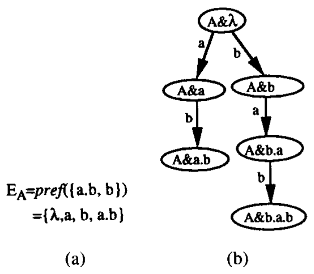


Fig. 8. (a) $E_A$ set; (b) the $s$-tree of the state A.

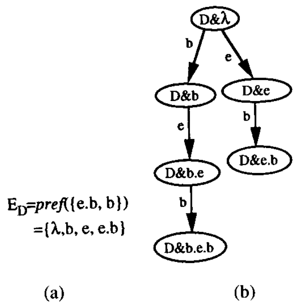


Fig. 9. (a) $E_D$ set; (b) the $s$-tree of the state D.

inputs in any e-sequence at $S$ are consumed in the order of the inputs in the e-sequence. Therefore, an e-sequence at $S$ must be a subsequence of an input sequence along a direct path starting from $S$ in the save-affected-graph. Consequently, $Z_S$ contains all maximal e-sequences at $S$; and thus $E_S$ is $pref(Z_S)$.

For the example shown in Figure 5, both the states A and D satisfy the simple save-affected-graph assumption. Then we derive $Z_A = \{a.b, b\}$ and $Z_D = \{e.b, b\}$ from the save-affected-graphs shown in Figures 6(c) and 7(c), respectively. Therefore, $E_A = pref(\{a.b, b\}) = \{\lambda, a, b, a.b\}$ and $E_D = pref(\{e.b, b\}) = \{\lambda, b, e, e.b\}$.

For the case that the simple save-affected-graph assumption is not satisfied, we present in Appendix III an algorithm for finding $E_S$ provided the set $E_S$ is finite; however, that algorithm is less efficient than the one presented above.

**Constructing *s*-Trees:**

We present in this section an algorithm for constructing an SDL-graph, called $s$-tree, for a state $S_i$. The $s$-tree of $S_i$ intuitively serves as a relation for partitioning the $\mathfrak{G}1_i$—the set of all stable global states related to the state $S_i$ as mentioned before—into a finite set of equivalence classes.

*Algorithm 1:* Construction of the *s-tree (save-corresponding-tree)* of a given state $S$.

**Input:** An SDL-machine F, and a given state $S$.
**Output:** The $s$-tree of the state $S$.
**Condition of applicability:** $E_S$ is finite.
**Step 1:** Construct $E_S$, the set of all $e$-sequences at $S$.
**Step 2:** Build a tree initially containing only one unmarked node labeled $S$ & $\lambda$.
**Step 3:** If all leaves of the resulting tree have been marked, then stop with the resulting tree being the $s$-tree of the state $S$. Otherwise,

1) Find in the resulting tree a unmarked leaf node labeled $S$ & $x$, and mark the node.
2) For every $b$ in $saveset(S)$, if $E_S \cap proj(x) \neq E_S \cap proj(x.b)$, then create an unmarked child node of $S$ & $x$ with label $S$ & $x.b$ in the tree. Go to Step 3. □

For the example shown in Fig. 5, using the $E_A$ and $E_B$ derived in Section III-B.1 (shown in Figs. 8(a) and 9(a)), the algorithm constructs the $s$-trees of the states $A$ and $D$, shown in Figs. 8(b) and 9(b), respectively.

The $s$-tree of $S$ intuitively captures the following notion: Consider an SDL-machine F and a state $S \in K$ where $E_S$ is finite. For $x \in saveset(S)^*$, in the $s$-tree of the state $S$, let the node $S$ & $z$ be the end state of the execution path obtained by applying $x$ to the root state $S$ & $\lambda$ (note: an $s$-tree is an SDL-graph, representing an SDL-machine); then, the input sequence $z$ is a $u$-sequence of $x$ at $S$. Consequently, for $x, y \in saveset(S)^*, [S, x]$ is equivalent to $[S, y]$ if the two execution paths obtained by applying $x$ and $y$ to the state $S$ & $\lambda$, respectively, have the same end state. Therefore, with the help of the $s$-tree of $S$, the set of all stable global states related to the state $S$ is partitioned into a finite set of equivalence classes, each of which corresponds to a node in the $s$-tree of $S$.

In Algorithm 1, Step 3(2) is intuitively based on the following idea: Given a state $S \in K$, let an input sequence $a_1.a_2 \cdots a_{n-1}.a_n \in saveset(S)^*, n \geq 1$, and $x = a_1.a_2 \cdots a_{n-1}$ (note: if $n = 1$, let $x = \lambda$). If $E_S \cap proj(x) = E_S \cap proj(x.a_n)$, then the $a_n$ of $x$. $a_n$ is a $n$-input of $x \cdot a_n$ at the state $S$. Therefore, there is no need to remember the $a_n$. In this case, $x$ is a $u$-sequence of $x.a_n$ at $S$.

We note that the condition of applicability of Algorithm 1 is decidable and can be determined using Algorithm 5 given in Appendix III.

We show that Algorithm 1 will terminate after a finite number of steps. Let an integer $M$ be $|E_S|(|E_S|$ is the number of elements of $E_S)$. We first argue that the $s$-tree of $S$ does not have any path from the root that is longer than $M-1$. Assume to the contrary that there exists a path $p$ of length $M$, and that $x.b$ is an input sequence along $p$ with $x \in saveset(S)^*$ and $b \in saveset(S)$. Then, according to Step 3(2),

$E_S \cap proj(x) = E_S$ since $|x| + 1 = |E_S| = M(|x|$ is the length of $x)$. Therefore,

$E_S \cap proj(x) = E_S \cap proj(x.b) = E_S$ since $E_S \cap proj(x) \subseteq E_S \cap proj(x.b) \subseteq E_S$.

This implies that the $s$-tree cannot have the path $p$ of length $M$. This contradiction concludes that all paths in the $s$-tree are not longer than $M-1$, thus the $s$-tree is finite. Consequently, according to Step 3(1), all leaves of the tree will eventually be marked, and therefore the algorithm will terminate since the tree is finite.

**Equivalent Transformation:**

We present in this section an algorithm of transforming SDL-machines into equivalent FSM's with the help of $s$-trees.

We note that an $s$-tree is an SDL-graph, thus represents an SDL-machine. The nodes and edges in an $s$-tree are viewed as states and explicit transitions in the SDL-machine represented by the tree.

Given an SDL-machine F where $E_S$ is finite for every state $S \in K$, using $s$-trees, this algorithm derives an equivalent FSM $F'$ from the machine F intuitively in the following manner: 1) Initially let $F'$ be the portion of the SDL-machine that is obtained from F by deleting all save constructs. 2) Construct the $s$-trees of all states that have save constructs, and add all $s$-trees to $F'$ by merging every pair of the root state $S$ & $\lambda$ of an $s$-tree and the state $S$ of $F'$ to form a state $S$. Therefore, in this $F'$, a state $S$ & $z$ introduced from an $s$-tree is equivalent to every stable global state $[S, x]$ in F where $S$ & $z$ is the end state of the execution path obtained by applying $x$ to $S$ in $F'$. 3) For every state $S$ & $z$ in $F'$, for every input $a \in in(S)$, add a transition labeled $a/out(S, z.a)$ to a state $Q$ in $F'$ such that $[Q, \lambda]$ is equivalent to the stable global state $[S@z.a, queue(S, z.a)]$ in F. Therefore the resulting $F'$ is equivalent to the original machine F.

*Algorithm 2:* Equivalent transformation of SDL-machines to avoid save constructs.

```
Input : An SDL-machine F.
Output: An equivalent FSM F'.
Condition of applicability: For every state $S$ in $K$, $E_S$ is finite.
Step 1: Draw the portion of the SDL-machine that is obtained from F by deleting all save constructs.
Step 2: Assume that $\mathbb{E}$ represents $\{S | S \in K \ \& \ saveset(S) \neq \emptyset\}$. Let $\mathbb{E} = \{S_1, S_2, \cdots, S_m\}$.
    For every state $S \in \mathbb{E}$, draw the $s$-tree of $S$ using Algorithm 1. Rewrite all $s$-trees in SDL-graphical symbols. Let $i := 1$.
Step 3: If $i > m$, then go to Step 4. Otherwise, for every node $S_i$ & $x$ in the $s$-tree of the state $S_i$ except for the root $S_i$ & $\lambda$, and for every $a \in in(S_i)$, do the following:
    1) Let $S_j = S_i@x.a$. Find the node $Q$ in the resulting graph such that:
        a) if $S_j \in \mathbb{E}, Q$ is the end state $S_j$ & $z$ of the execution path in the $s$-tree of the state $S_j$ obtained by applying the input sequence $queue(S_i, x.a)$ to the state $S_j$ & $\lambda$.
        b) otherwise (i.e., $S_j \notin \mathbb{E}$), $Q$ is the state $S_j$ in the portion of graph created in Step 1.
    2) Create a transition from the node $S_i$ & $x$ to the node $Q$ with the label $a/out(S_i, x.a)$. Let $i := i + 1$. Go to Step 3.
Step 4: Rename every node $S$ & $\lambda$ by $S$ in the resulting graph. The resulting graph is the equivalent FSM F'. □
```

For the example shown in Fig. 5, from the $s$-trees of states A and D, we construct an equivalent FSM as shown in Fig. 10. In Fig. 10, Step 1 draws the portion that is not contained in the dashed block. For the state $A$ & $a$ in the $s$-tree of $A$ and the input $c$, Step 3(1) finds that the $Q$ is the state $D$, and Step 3(2) creates the transition from $A$ & $a$ to $D$ with the label $c/out(A, a.c)$ where $out(A, a.c) = x.z$. For the state $A$ & $a.b$ in the $s$-tree of $A$ and the input $c$, Step 3(1) finds that the $Q$ is the state $D$ & $b$, and Step 3(2) creates the transition from $A$ & $a.b$ to $D$ & $b$ with the label $c/out(A, a.b.c)$ where $out(A, a.b.c) = x.z$.

In Algorithm 2 only Step 3 may be repeated. It is performed for adding (a finite number of) outgoing transitions to the nodes of the $s$-trees. Since the number of nodes of the $s$-trees is finite, this algorithm terminates after a finite number of steps. The validity of the algorithm is given as follows.

*Theorem 2:* Algorithm 2 transforms SDL-machines into equivalent FSM's under the condition of applicability. Proof: See Appendix I.

*Theorem 3:* For a given SDL-machine where every explicit transition has at least one nonempty output, there exists an equivalent FSM if and only if the condition of applicability of Algorithm 2 is satisfied. Proof: See Appendix I.

This theorem shows that the applicability condition of Algorithm 2 is a necessary and sufficient condition for the existence of an equivalent FSM for a given SDL-machine where every explicit transition has at least one nonempty output.

## IV. Test Design

We give in this section a method for test selection from SDL-machines that is based on our equivalent transformation algorithm. We present a fault model that includes output faults and transfer faults that are usual for FSM's (i.e., the output corresponding to a transition is erroneous or there is a fault in the next state reached by a transition [8]–[10]) and the save faults that are specific to SDL-machines. We also discuss the fault coverage of the test cases derived by our method under the given fault model.

Let SP be a specification and IUT its implementation. Assume that they have the same $I$ and $O$. The fault types are defined as follows:

1) *Output fault:* We say that IUT has output faults if 1) IUT is not equivalent to SP and 2) SP can be obtained from IUT by modifying the outputs of one or more transitions in IUT.
2) *Transfer fault:* We say that IUT has transfer faults if 1) IUT is not equivalent to SP and 2) SP can be obtained from IUT by modifying the end states of one or more transitions in IUT.
3) *Save fault:* We say that IUT has save faults if 1) IUT is not equivalent to SP and 2) SP can be obtained from IUT by modifying the labels (i.e., inputs) in one or more save constructs in IUT.
4) *Hybrid fault:* We say that IUT has hybrid faults if 1) IUT is not equivalent to SP and 2) SP can be obtained from IUT by changing the outputs and/or the end states of one or more transitions, and/or by modifying the labels in one or more save constructs, in IUT.

For SDL-machines that satisfy the condition of applicability of Algorithm 2, we use the following procedure to generate test suites.

*Test generation procedure for a given SDL-machine:*

**Step 1:** Transform the given SDL-machine into an equivalent FSM using Algorithm 2.

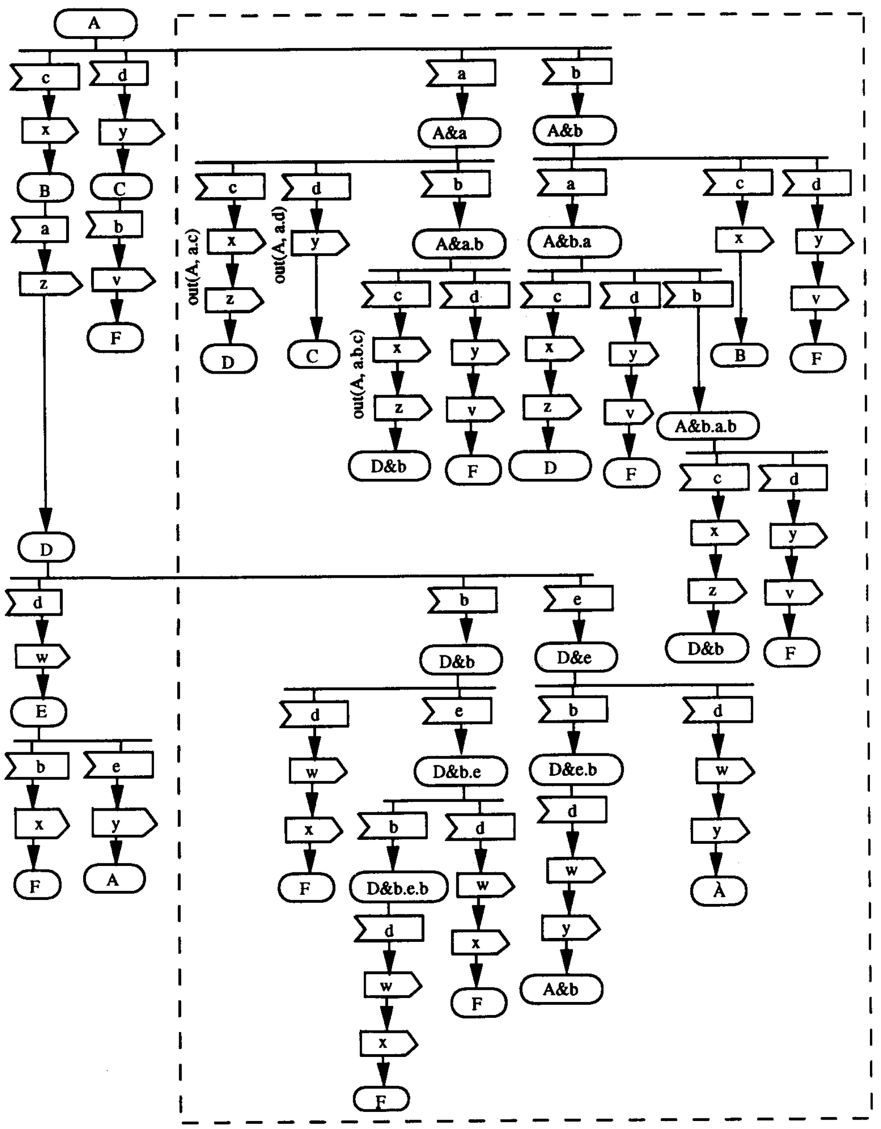


Fig. 10. An equivalent FSM of the SDL-machine of Fig. 5.

**Step 2:** Ignore the existence of the input queue of the resulting FSM, and generate test cases from the machine using one of the test suite development methods for finite state machines, such as the W-method [9], Wp-method [10], UIO-method [11] or transition tour [12]. □

Under the fault model given before, which implies that we assume that no faults other than those in the fault model can occur, the fault coverage is given as follows.

*The fault coverage of the test suite:* If the methods used in Step 2 of the test generation procedure outlined earlier can ensure the complete fault coverage for FSM's under the fault model which only assumes the mixed output and transfer faults, the test suite can detect any fault specified in the fault model (i.e., all the four types of faults). This is because save faults can be modeled as output faults and transfer faults in the transformed equivalent FSM's. This implies complete fault coverage.

For SDL-machines that do not satisfy the condition of applicability of Algorithm 2, we use the heuristic approach given in [5] to transform a given SDL-machine to an FSM that is an approximation of the original SDL-machine. Test case selection is based on the resulting FSM. Therefore, the equivalence relation between specifications and implementations cannot be fully guaranteed, and complete fault coverage cannot be guaranteed either.

We note that in our experience most SDL-machines derived from practical SDL specifications have equivalent FSM's. Therefore, our approach results in good fault coverage for most practical applications.

## V. Conclusion

The signal save construct is one of the features distinguishing SDL from traditional high-level specification and programming languages. However, this feature increases the difficulties of testing SDL-specified software. We present a testing approach consisting of the following three phases: SDL specifications are first abstracted into finite state machines with save constructs, called *SDL-machines*; the resulting SDL-machines are then transformed to equivalent FSM's if this is possible; and, finally, test cases are selected from the resulting FSM's. We concentrate on the second phase and come up with an equivalent transformation algorithm for this phase, since there are existing methods for the first and third phases. The applicability condition of this algorithm is a necessary and sufficient condition for the existence of an equivalent FSM, for a given SDL-machine where every explicit transition has at least one nonempty output.

In the area of verification for SDL-specified software, the specifications usually need to be abstracted into FSM's for verifying a so-called deadlock-free property. This can be done by first abstracting SDL specifications into SDL-machines and then applying this transformation method to obtain equivalent FSM's. Therefore, the equivalent transformation method could be useful in that area as well.

## Appendix I
## Proofs of Validation of the Method

In order to prove Theorems 2 and 3, we need several lemmas. For the sake of convenience, we assume in the following that all SDL-machines being discussed are denoted by $(K, I, O, saveset, T, S_0)$ unless we specify them explicitly.

*Lemma 1:* Given a state $S \in K$ and an input sequence $x \in saveset(S)^*$, let $z$ be a $u$-sequence of $x$ at $S$. Then, $\forall y \in I^*$ (out$(S, x.y)$ = out$(S, z.y)$ & $S@x.y = S@z.y$ ).

*Proof:* From the definition of $u$-sequences, $z$ is obtained from $x$ only by eliminating zero or more $n$-inputs of $x$ at the state $S$. For every $y \in I^*$, apply $x.y$ to $S$; then any of such $n$-inputs when consumed does not stimulate any output and only causes a self-loop at a state, since $n$-inputs when consumed are consumed by implied transitions. Therefore, the same sequence of explicit transitions will be consumed when $x.y$ and $z.y$ are applied to $S$, respectively. Thus the lemma holds. □

To prove the next lemma, we need the following concept.

*Definition:* Comparison of two strings of integers.

Given two strings of integers $k_1.k_2 \cdots k_n$ and $l_1.l_2 \cdots l_n$ of the same length where $k_1 < k_2 < \cdots < k_n$ and $l_1 < l_2 < \cdots < l_n$, we say that $k_1.k_2 \cdots k_n$ is smaller than $l_1.l_2 \cdots l_n$ *if*

$$\exists i(k_1.k_2 \cdots k_{i-1} = l_1.l_2 \cdots l_{i-1} \ \& \ k_i < l_i).$$

□

*Lemma 2:* Given a state $S \in K$, let an input sequence $a_1.a_2 \cdots a_{n-1}.a_n \in saveset(S)^*, n \geq 1$, and $x = a_1.a_2 \cdots a_{n-1}$ (note: if $n = 1$, let $x = \lambda$). If $E_S \cap proj(x) = E_S \cap proj(x.a_n)$, then the $a_n$ of $x.a_n$ is a $n$-input of $x.a_n$ at the state $S$ (note: in this case, $x$ is a $u$-sequence of $x.a_n$ at $S$.).

*Proof:*

**Part I:** We note the following fact: Given a state $S \in K$, consider an input sequence $a_1.a_2 \cdots a_n \in saveset(S)^*$ and an input sequence $w \in I^*$. Assume that $a_{k1}.a_{k2} \cdots a_{ki}$ is obtained from $a_1.a_2 \cdots a_n$ by eliminating all inputs that are consumed by implied transitions when $a_1.a_2 \cdots a_n.w$ is applied to $S$. Then there does not exist any $a_{l1}.a_{l2} \cdots a_{li}$ such that $l_1.l_2 \cdots l_i$ is smaller than $k_1.k_2 \cdots k_i$ and $a_{l1}.a_{l2} \cdots a_{li} = a_{k1}.a_{k2} \cdots a_{ki}$.

**Part II:** We now prove the lemma.

(1) $E_S \cap proj(x) = E_S \cap proj(x.a_n)$
  assumption

(2) $\exists w \in I^*$ (the $a_n$ in $x.a_n.w$ will be consumed by an explicit transition when $x.a_n.w$ is applied to the state $S$ )
  assuming the contrary of the lemma

considering such a $w$ in the following

(3) derive the sequence $a_{k1}.a_{k2} \cdots a_{ki}.a_n$ from $x.a_n$ by deleting all inputs in $x.a_n$ that will be consumed by implied transitions when $x.a_n.w$ is applied to the state $S$

(4) $a_{k1}.a_{k2} \cdots a_{ki}.a_n \in E_S \cap proj(x.a_n)$
  (3)

(5) $a_{k1}.a_{k2} \cdots a_{ki}.a_n \in E_S \cap proj(x)$
  (1) & (4)

(6) there must be a $a_{l1}.a_{l2} \cdots a_{li}.a_m$ such that
 i) $l_1.l_2 \cdots l_i.m$ is smaller than $k_1.k_2 \cdots k_i.n$, and
 ii) $a_{l1}.a_{l2} \cdots a_{li}.a_m = a_{k1}.a_{k2} \cdots a_{ki}.a_n$, and
 iii) $m < n$
  (5)

(7) (6) is not true
  Part I

(8) The lemma holds
  (2) causes the contradiction between (6) and (7).

□

*Lemma 3:* Given a state $S \in K$, an input sequence $x.a \in saveset(S)^*$ and $a \in saveset(S)$, if an input sequence $z$ is a $u$-sequence of $x$ at $S$, then $z.a$ is a $u$-sequence of $x.a$ at $S$.

*Proof:*

1) $z$ is a u-sequence of $x$ at the state $S$
  assumption
2) all n-inputs of $x$ at $S$ are n-inputs of $x.a$ at $S$
  definition of n-inputs
3) $z.a$ is obtained from $x.a$ by eliminating zero or more n-inputs of $x.a$ at $S$
  1) & 2)
4) $z.a$ is a u-sequence of $x.a$ at $S$
  3) & definition of u-sequences.

□

We use in the following the terminology of SDL-machines for $s$-trees, since an $s$-tree is an SDL-graph representing an SDL-machine. The nodes and edges in an $s$-tree are viewed as states and transitions in the SDL-machine represented by the tree.

*Lemma 4:* For a state $S \in K$ where all $e$-sequences at $S$ are of finite length, for $x \in saveset(S)^*$, in the $s$-tree of the state $S$, let the node $S$ & $z$ be the end state of the execution path obtained by applying $x$ to the state $S$ & $\lambda$ (the root of the tree). Then, the input sequence $z$ is a $u$-sequence of $x$ at $S$.

*Proof:* We prove the lemma by induction on the length of $x$.

*Induction Hypothesis:* for $x \in saveset(S)^*$, we assume that $S$ & $z$ is the end state of the execution path obtained by applying $x$ to the state $S$ & $\lambda$. Then the input sequence $z$ is a $u$-sequence of $x$ at $S$.

*Induction Base:* $x = \lambda$. $S$ & $\lambda$ is the end state of the execution path obtained by applying $x$ to the state $S$ & $\lambda$. In this case, since $\lambda$ is a $u$-sequence of $x$ at the state $S$, the lemma holds.

*Induction Step:* for $x.a \in saveset(S)^*$, we assume that $S$ & $y$ is the end state of the execution path obtained by applying $x.a$ to the state $S$ & $\lambda$. We argue that the input sequence $y$ is a $u$-sequence of $x.a$ at the state $S$ as follows:

1) The Induction Hypothesis given before
  assumption
2) the $y$ is either the $z$ or the $z.a$
  1)
3) $z.a$ is a u-sequence of $x.a$ at the state $S$
  1) & Lemma 3
4) the $y$ is the $z$
  assumption
5) $E_S \cap proj(z) = E_S \cap proj(z.a)$
  4) & Algorithm 1
6) $z$ is a u-sequence of $z.a$ at the state $S$
  5) & Lemma 2
7) $z$ is a u-sequence of $x.a$ at the state $S$
  6) & definition of u-sequence
8) the $y$ is a u-sequence of $x.a$ at the states
  2) & 3) & "4) $\Longrightarrow$ 7)"

□

For the sake of convenience, we assume the following:

1) F$(K, I, O, saveset, T, S_0)$ is an SDL-machine such that, for every state $S \in K, E_S$ is finite.
2) F$'(K', I, O, saveset', T', S_0)$ is the resulting machine obtained from F using Algorithm 2.
3) In contrast to the @ and *out* functions for F, @$'$ and *out*$'$ refer to the corresponding functions for F$'$, respectively.
4) $\mathbb{E} = \{S | S \in K \ \& \ saveset(S) \neq \emptyset\}$.

It is easy to see: 1) for every state $Q \in K', saveset'(Q) = \emptyset$. 2) $K \subseteq K'$ 3) all explicit transitions of F are explicit transitions of F$'$.

*Lemma 5:* Let $S \in K$ and $a \in I$. Let $S$ & $z$ be a state in the $s$-tree of the state $S$ for the machine F. The following statements hold:

(a) $out(S, z.a) = out'(S, z.a)$;
(b) $S@'z.a = (S@z.a)@'queue(S, z.a)$.

*Proof:*

**Case I:** $a \notin in(S)$.

1) $a \notin$ in$(S)$
  assumption
2) $out(S, z.a) = \lambda$ and $out'(S, z.a) = \lambda$
  1) & $z \in saveset(S)$
3) *Statement a) holds (i.e.,* $out(S, z.a) = out'(S, z.a)$*)*
  2)
4) $S@z.a = S$
  1) &$z \in saveset(S)$
5) $(S@z.a)@'queue(S, z.a) = S@'queue(S, z.a)$
  4)
  $= S@'z.a$
  1) & $z \in saveset(S)$ & the s-tree of $S$
6) *Statement b) holds*
  5)

**Case II:** $a \in in(S)$.

1) $a \in in(S)$
  assumption
2) In $F'$, when $z$ is applied to the state $S$, the machine will reach the state $S$ & $z$ without any output produced
  only a path in s-tree is executed
3) In F$'$, when $a$ is applied to the state $S$ & $z$, the machine will produce the output sequence $out(S, z.a)$
  1) & Step 3 of Algorithm 2
4) *The statement a) holds (i.e.,* $out(S, z.a) = out'(S, z.a)$*)*
  2) & 3)
5) Let $P = S@z.a$
  In F$'$, when a is applied to the state $S\&z$, the machine will reach the state $Q$ where $Q$ is decided as follows:
  i) if P$\in \mathbb{E}$, then $Q$ is the end state $P\&y$ of the execution path obtained by applying the input sequence $queue(S, z.a)$ to the state $P$ (i.e., the node $P\&\lambda$ in the s-tree). Therefore, $Q = (S@z.a)@'queue(S, z.a)$.
  1) & Step 3 of Algorithm 2
  ii) if $P \in \mathbb{E}$, then $Q$ is $P$. In this case, $P = S@z.a, queue(S, z.a) = \lambda$.
  Therefore, still, $Q = (S@z.a)@'queue(S, z.a)$.
  1) & Step 3 of Algorithm 2
6) *The statement b) holds*
  (i.e.,$S@'z.a = (S@z.a)@'queue(S, z.a)$)
  2) & 5).□

*Lemma 6:* Consider a state $S \in K$ and an input $a \in I$. Let $S\&z$ be a state in the $s$-tree of $S$ for the machine F. Then, for any $w \in I^*$, $out'(S@z.a, queue(S, z.a).w) = out'((S@z.a)@'queue(S, z.a), w)$.

*Proof:*

1) $queue(S, z.a) \in saveset(S@z.a)^*$
  definitions of queue and @
2) $\forall P \in K \forall x \in saveset(P)^*(out'(P, x) = \lambda)$
  in F', when $x$ is applied to the $P$, only a path within the s-tree of $P$ will be executed without producing any output.
3) $out'(S@z.a, queue(S, z.a)) = \lambda$
  1) & 2)
4) $\forall\ y, w \in I^*(out'(S, y.w) = out'(S, y).out'(S@'y, w))$
  $\forall Q \in K'(saveset'(Q) = \emptyset)$
5) for $w \in I^*, out'(S@z.a, queue(S, z.a).w)$

TABLE II
ADDITIONAL NOTATIONS FOR SDL-MACHINES

| Notations | Meaning |
|---|---|
| $S_1 - a \rightarrow S_2$ | there exists $w \in O^*$ $S_1 - a/w \rightarrow S_2$ |
| $S_1 = b_1 \ldots b_{m-1} \Rightarrow S_m$ | there exist $S_2, \ldots, S_{m-1} \in K$ such that $S_1 - b_1 \rightarrow S_2 - b_2 \rightarrow S_3, \ldots, S_{m-1} - b_{m-1} \rightarrow S_m$ |

$= out'(S@z.a, queue(S, z.a)).$
$\quad out'((S@z.a)@'queue(S, z.a), w)$
4)
$= out'((S@z.a)@'queue(S, z.a), w)$
3).

□

*Theorem 2:* Algorithm 2 transforms SDL-machines into equivalent FSM's under the conditions of applicability. (That is, $\forall S \in K \forall x \in I^*(out(S, x) = out'(S, x))$.)

*Proof:* The proof is straightforward for the situation that none of the inputs of $x$ is in $in(S)$. We now prove the lemma for the case that there exists at least one input in $x$ belonging to $in(S)$, by induction on the length of $x$.

*Induction Hypothesis:* for a positive integer $i$, $\forall S \in K \forall x \in I^*(|x| < i \Longrightarrow out(S, x) = out'(S, x))$.

*Induction Base:* $|x| = 1$. In this case, $x \in in(S)$. Therefore, the lemma holds because the machine F′ contains all explicit transitions of F.

*Induction Step:* Assume $x \in I^*$ and $|x| = i > 1$. Let $x = x_1.a.v$ where $a \in in(S), x_1 \in saveset(S)^*$, and $v \in I^*$. In the $s$-tree of the state $S$, let the node $S$ & $z$ be the end state of the execution path obtained by applying the $x_1$ to the state $S$ & $\lambda$. We have

$out(S, x) = out(S, x_1.a.v)$
$x = x_1.a.v$
$= out(S, z.a.v)$
Lemmas 4 and 1
$= out(S, z.a).out(S@z.a, queue(S, z.a).v)$
definition of *out*
$= out'(S, z.a).out(S@z.a, queue(S, z.a).v)$
Lemma 5(a)
$= out'(S, z.a).out'(S@z.a, queue(S, z.a).v)$
Hypothesis of induction
$= out'(S, z.a).out'((S@z.a)@'queue(S, z.a), v)$
Lemma 6
$= out'(S, z.a).out'(S@'z.a, v)$
Lemma 5(b)
$= out'(S, z.a.v)$
definition of *out′*& $\forall Q \in K'(saveset'(Q) = \emptyset)$
$= out'(S, x_1.a.v)$
the node $S\&z$ is the end state of the execution path in the tree when the $x_1$ is applied to the state $S\&\lambda$
$= out'(S, x) = x_1.a.v.$

□

For the sake of convenience, we introduce in the following additional notations for SDL-machines.

We require the following concept for proving Theorem 3.

*Definition:* Save-affected-path.

Given a state $S_1 \in K$, a path in an SDL-graph is called a *save-affected-path* from $S_1$ if

1) the path starts from the state $S_1$, and
2) if the path is represented in the following form, called a *normal form* (note: this is a unique form):

$$S_1 - b_1 \rightarrow Q_1 = x_1 \Rightarrow S_2 - b_2 \rightarrow Q_2 = x_2 \Rightarrow$$
$$S_3 \cdots S_m - b_m \rightarrow Q_m = x_m \Rightarrow S_{m+1}$$

where $m \geq 1$, $b_i \notin saveset(S_1)$, and $x_i \in saveset(S_1)^*$ for $i = 1, 2, \cdots, m$,
then $x_i \in (\bigcap_{k=1}^{i}(saveset(S_k))^*$ for $i = 1, 2, \cdots, m$, and $x_m \neq \lambda$. □

*Lemma 7:* Given a state $S$, all $e$-sequences of the state $S$ are of finite length if and only if none of the save-affected-paths of $S$ contain any directed cycle that has at least one transition with an input in $saveset(S)$.

*Proof:* This theorem is evident from the definitions of $e$-sequences and save-affected-paths. □

According to Lemma 7, checking the conditions of applicability of Algorithms 1 and 2 is reduced to checking whether, for every state $S$, all save-affected-paths of $S$ do not contain any directed cycle that has at least one transition with an input in $saveset(S)$.

*Theorem 3:* For a given SDL-machine where every explicit transition has at least one nonempty output, there exists an equivalent FSM if and only if the condition of applicability of Algorithm 2 is satisfied.

*Proof: (I. Sufficiency):* Theorem 2 proves that if the condition of applicability of Algorithm 2 is satisfied, then there exists an equivalent FSM.

*(II. Necessity):* We prove in the following that the condition is also necessary. Assume the contrary that given an SDL-machine F, there exists $S$ in $K$ such that not all $e$-sequences of the state $S$ are of finite length. According to Lemma 7, there must exist a state $S$ in $K$ and a save-affected-path from $S$ such that the path contains a directed cycle which has at least one transition with an input in $saveset(S)$. In this save-affected-path, we find the shortest path $p$ from $S$ to the cycle, and we assume that

i) $x_1$ is the input sequence along the path $p$,
ii) $Q$ is the end state of the path $p$,
iii) $x_2$ is the input sequence along the cycle from $Q$ to itself,
iv) $y_1$ is obtained by eliminating all the inputs of $x_1$ that do not belong to $saveset(S)$,
v) $z_1$ is obtained by eliminating all the inputs of $x_1$ that belong to $saveset(S)$,
vi) $y_2$ is obtained by eliminating all the inputs of $x_2$ that do not belong to $saveset(S)$,
vii) $z_2$ is obtained by eliminating all the inputs of $x_2$ that belong to $saveset(S)$,
viii) since F is initially connected, there must exist a path in F from the initial state $S_0$ to the $S$; let $x_0$ be the input sequence along such a path.

*Part A:* We first argue that none of the stable global states $[S, y_1.y_2], [S, y_1.y_2^2], \cdots, [S, y_1.y_2^i], \cdots$ are equivalent.

Consider two stable global states $[S, y_1.y_2^i]$ and $[S, y_1.y_2^{i+k}]$, for $i = 1,2,3,\cdots$ and $k = 1,2,3,\cdots$

1) given $P \in K$, and $x, y \in I^*$, if $out(P, x.z) \neq out(P, y.z)$ for some $z \in I^*$, then the two global states $[P, x]$ and $[P, y]$ are not equivalent

  definition of equivalence

2) let $x_2 = v.a.w$ where $a \in saveset(S)$ and no input of $v$ is in $saveset(S)$

  definition

3) if $v = \lambda$, then for $z_1.z_2^i \in I^*$,
out$(S, y_1.y_2^{i+k}.z_1.z_2^i) \neq out(S, y_1.y_2^i.z_1.z_2^i)$.
(note: out$(S, y_1.y_2^{i+k}.z_1.z_2^i)$
$= out(S, y_1.y_2^i.z_1.z_2^i).out(Q, y_2^k)$ )
if $v \neq \lambda$, then for $z_1.z_2^i.v \in I^*$,
$out(S, y_1.y_2^{i+k}.z_1.z_2^i.v) \neq out(S, y_1.y_2^i.z_1.z_2^i.v)$
(note:$out(S, y_1.y_2^{i+k}.z_1.z_2^i.v)$
$= out(S, y_1.y_2^i.z_1.z_2^i.v).out(Q@v, y_2^k)$)

  definition of out & 2) &
  "every explicit transition of F has at least one nonempty output" &
  "the cycle has at least one transition with an input in $saveset(S)$"

4) $[S, y_1.y_2^i]$ and $[S, y_1.y_2^{i+k}]$ are not equivalent

  1) & 3)

5) none of the stable global states $[S, y_1.y_2], [S, y_1.y_2^2], \cdots, [S, y_1.y_2^i], \cdots$ are equivalent

  4)

*Part B:* We argue that there does not exist any equivalent FSM for the given SDL-machine F. According to Part A, none of the stable global states $[S, y_1.y_2], [S, y_1.y_2^2], \cdots, [S, y_1.y_2^i\ ], \cdots$, are equivalent. Consequently, for any SDL-machine equivalent to F, the input sequences $x_0.y_1.y_2, x_0.y_1.y_2^2, \cdots, x_0.y_1.y_2^i, \cdots$ must lead the machine from its initial global state to a set of stable global states $T_1, T_2, \cdots, T_i, \cdots$ that are equivalent to $[S, y_1.y_2], [S, y_1.y_2^2], \cdots, [S, y_1.\ y_2^i], \cdots$, respectively. None of $T_1, T_2, \cdots, T_i, \cdots$ are equivalent. Thus, such an equivalent machine must have an infinite number of stable global states since $i$ can take any positive integer value. An FSM only has a finite number of stable global states since it does not have any save construct. Therefore, there does not exist any equivalent FSM for the given SDL-machine F.

*Part C:* From Part B, if the give SDL-machine F does not satisfy the condition of applicability of Algorithm 2, then there does not exist any equivalent FSM for F. Therefore, if there exists an equivalent FSM for the give SDL-machine F, then F must satisfy the condition of applicability of Algorithm 2. □

## Appendix II
## Algorithms for Constructing Save-Graphs and Save-Affected-Graphs

Given a state $S$ in an SDL-machine and a set of inputs $\mathbb{Z} \subseteq saveset(S)$, we define a so-called *save-graph* of $S$ with respect to $\mathbb{Z}$ that intuitively captures the following notion: For every input sequence $x \in \mathbb{Z}^*$ and an input $a \in in(S)$, the inputs of $x.a$ can only be consumed by the transitions in the save-graph of $S$ with respect to $\mathbb{Z}$. We formally give a constructive definition of save-graphs by the following algorithm. Note that we always present save-graphs in the form of SDL-graphs with all outputs and save constructs omitted.

*Algorithm 3:* Construction of the save-graph of a given state.

**Input:** An SDL-machine F, a given state $S$, and a set of inputs $\mathbb{Z} \subseteq saveset(S)$.
**Output:** The save-graph of $S$ with respect to $\mathbb{Z}$.
**Step 1:** Let all transitions in the SDL-machine F be initially unmarked. Mark the state $S$. Let $G$ always represent the marked portion of F (thus $G$ initially contains only the state $S$).
**Step 2:** Find in F all transitions starting from $S$ such that the end state of such a transition has an outgoing transition with an input in $\mathbb{Z}$. Modify $G$ by marking these transitions and their end states.
Let $V$ be the set of all the end states of the marked transitions resulting from this step.
**Step 3:** Find in F all transitions starting from $S$ such that, for the end state $Q$ of such a transition, $\mathbb{Z} \cap saveset(Q) \neq \emptyset$. Modify $G$ by marking these transitions and their end states.
**Step 4:** Find all transitions with the inputs in $\mathbb{Z}$, each of which can be reached from a state in $V$, along a directed path $p$ in F with the inputs of all transitions in the path $p$ belonging to $\mathbb{Z}$. Modify $G$ by marking these transitions and their end states.
**Step 5:** Stop. The resulting $G$ is the save-graph of $S$ with respect to $\mathbb{Z}$. The root of the save-graph is the state $S$. □

Figs. 6 and 7 show the examples of the save-graphs.

In the following, an *elementary path* refers to a path where all edges are distinct.

*Algorithm 4:* Construction of the save-affected-graph of a given state.

**Input:** An SDL-machine F, and a given state $S$.
**Output:** The save-affected-graph of $S$.
**Step 1:** Let all transitions in the SDL-machine F be initially unmarked. Then mark all transitions and their adjacent states in F that belong to the save-graph of $S$ with respect to $saveset(S)$. Let $G$ always represent the marked portion of F.
**Step 2:** Find a state $Q$ in $G$ such that

i) there exists an elementary path from $S$ to $Q$; and if the path is presented in the following normal form:

$$S = S_1 - b_1 \rightarrow Q_1 = x_1$$
$$\Rightarrow S_2 - b_2 \rightarrow Q_2 = x_2$$
$$\Rightarrow S_3 \cdots S_m - b_m \rightarrow Q_m = x_m$$
$$\Rightarrow S_{m+1} = Q$$

then $x_i \in (\bigcap_{k=1}^{i}(saveset(S_k)))^*$ for $i = 1, 2, \cdots, m$.
(Note: by the definition of normal form, $m \geq$

$1, b_i \notin saveset(S_1)$, and $x_i \in saveset(S_1)^*$ for $i = 1, 2, \cdots, m$)

ii) let $\mathbb{Z} = \bigcap_{k=1}^{m+1} saveset(S_k)$, then $\mathbb{Z} \neq \emptyset$ and the save-graph of $Q$ with respect to $\mathbb{Z}$ is not a subgraph of $G$ (i.e., there is at least a unmarked transition in the save-graph.).

**Step 3:** If a state $Q$ has been found in Step 2, then 1) modify $G$ by marking all transitions and their adjacent states in F that are contained in the save-graph of $Q$ with respect to $\mathbb{Z}$, and 2) go to Step 2. Otherwise, go to next step.

**Step 4:** Find a transition $t$ in $G$ such that 1) the end state of the transition $t$ does not have any outgoing transition in $G$, and 2) the input of $t \notin saveset(S)$.

**Step 5:** If a transition $t$ has been found in Step 4, then modify $G$ by unmarking the $t$ in F and go to Step 4. Otherwise, stop; and the resulting G is the save-affected-graph of $S$ with its root being $S$. □

In this algorithm, each application of Steps 2 and 3 marks at least one transition. According to Step 2(ii), if there is no unmarked transition in F, then no $Q$ can be found in Step 2; thus Step 3 cannot be applied. Therefore, Steps 2 and 3 can be applied only a finite number of times, since F is a finite graph. Steps 4 and 5 are for making the save-affected-graph minimal. It is straightforward to prove that Steps 4 and 5 can be applied only a finite number of times also. Consequently this algorithm terminates after a finite number of steps.

## Appendix III
## A General Algorithm for Finding All $E$-Sequences

We present in this section an algorithm that, for a given SDL-machine and a state $S$, finds $E_S$ when $E_S$ is finite, or reports "$E_S$ is infinite" when $E_S$ is not finite. For ease of understanding, the given algorithm has not been optimized.

*Definition:* Corresponding-e-sequence.

Given a state $S$, for a save-affected-path $p$ from $S$, an input sequence $x$ is called a *corresponding-e-sequence* of $p$ if $\exists y \in I^*$. (All inputs in $x.y$ will be consumed by explicit transitions in the path $p$ when $x.y$ is applied to the state $S$.). For a save-affected-path $p$ from $S$, the set of all corresponding-e-sequences of the path p is written $ce(p)$. (Note: $\lambda \in ce(p)$.) □

This concept is based on the following intuition: Given a state $S$, if $E_S$ is finite, then for every e-sequence $x$ at $S$, there must be an elementary save-affected-path $p$ from $S$ such that $x$ is a corresponding-e-sequence of $p$.

In the following algorithm, we say that a save-affected-path $p$ from a state $S$ is maximal if there is no other save-affected-path $p_1$ from the state $S$ such that $p$ is a prefix (subpath) of $p_1$.

*Algorithm 5:* Construction of all e-sequences of a given state.

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

**Output:** 1. $E_S$, the set of all e-sequences at $S$ if $E_S$ is finite. 2. reporting "$E_S$ is infinite" if $E_S$ is infinite.

**Step 1:** Find the set $\mathbb{P}$ that contains all maximal elementary save-affected-paths from the state $S$ as follows: Assume that $M$ is the number of all explicit transitions in F, and that $\mathbb{P}1$ is the set of all elementary directed paths from $S$ in F (note: $\mathbb{P}$ is a finite set.).

**1)** Check the paths in $\mathbb{P}1$ one by one, and find in $\mathbb{P}1$ all save-affected-paths from the state $S$ that are either maximal or of the length $M$. Let $\mathbb{P}2$ be the set of these save-affected-paths.

2) If none of the paths in $\mathbb{P}2$ contains any directed cycle that has at least one transition with an input in $saveset(S)$, then let $\mathbb{P} = \mathbb{P}2$ and go to Step 2. Otherwise, stop and report "$E_S$ is infinite."

**Step 2:** For every $p \in \mathbb{P}$, construct $ce(p)$ as follows: For the path $p$ in $\mathbb{P}$, assume

i) the path is represented in the following normal form (note: this is a unique form):

$$\begin{aligned} S &= S_1 - b_1 \rightarrow Q_1 = x_1 \\ &\Rightarrow S_2 - b_2 \rightarrow Q_2 = x_2 \\ &\Rightarrow S_3 \cdots S_m - b_m \rightarrow Q_m = x_m \\ &\Rightarrow S_{m+1} \end{aligned}$$

where $m \geq 1$, $b_i \notin saveset(S_1)$ and $x_i \in saveset(S_1)^*$ for $i = 1, 2, \cdots, m$.

ii) $Np$ is the number of all transitions with inputs in $saveset(S)$ in the path $p$ (i.e., $Np = \sum_{i=1}^{m} |x_i|$).

iii) $saveset(S)^{Np}$ is the set of all input sequences of length $Np$ over $saveset(S)$.

Then, do the following:

- Construct a set $\mathbb{B}_p$ in the following way: Initially, let $\mathbb{B}_p = \emptyset$. Check every input sequence $x$ in $saveset(S)^{Np}$. If all inputs of $x$ will be consumed by explicit transitions in the path $p$ when $x.b_1 \cdots b_m$ is applied to $S$, then put $x$ into $\mathbb{B}_p$. (Note: the set $\mathbb{B}_p$ contains all $e$-sequences at the state $S$ of length $Np$ that can be consumed along the path $p$.)
- Construct $ce(p) = pref(\mathbb{B}_p)$.

**Step 3:** Construct $E_S = \bigcup_{p \in P} ce(p)$. □

We now explain the algorithm. I) We first argue that Step 1 finds all maximal elementary save-affected-paths from the state $S$ when $E_S$ is finite. If none of the save-affected-paths from $S$ contain any directed cycle that has at least one transition with an input in $saveset(S)$ (i.e., $E_S$ is finite from Lemma 7), then none of the maximal elementary save-affected-paths from $S$ are longer than $M$; therefore, we can find all maximal elementary save-affected-paths from $S$ by checking the set of all elementary directed paths from $S$ in F, as described in Step 1(1).

If there exists a save-affected-path from $S$ containing a directed cycle that has at least one transition with an input in *saveset* $(S)$ (i.e., $E_S$ is infinite from Lemma 7), then there must exist an elementary save-affected-path of the state $S$ containing a directed cycle that has at least one transition with an input in *saveset* $(S)$; this is checked up in Step 1(2) with

"$E_S$ is infinite" reported. If the algorithm does not stop in Step 1(2), then $E_S$ is finite according to Lemma 7.

II) It is easy to see that Step 2 constructs $ce(p)$ for a save-affected-path $p$ from the state $S$ by checking exhaustively. Therefore, Step 3 constructs $E_S$.

Using the concept of the save-affected-graph, Algorithm 5 can be optimized by changing the statement "$\mathbb{P}1$ is the set of all elementary directed paths from $S$ in F" in Step 1, into "$\mathbb{P}1$ is the set of all elementary directed paths from $S$ in the save-affected-graph of $S$." Algorithm 5 could be further optimized; however, these opitimizations are not discussed in this paper.

## ACKNOWLEDGMENT

The authors would like to thank Xiaoyu Song and Huishan Zhou for reading our paper and giving us their useful comments.

## REFERENCES

[1] Roberto Saracco, J. R. W. Smith, and Rick Reed, *Telecommunications Systems Engineering Using SDL.* New York: North-Holland, 1989.

[2] Dieter Hogrefe, "Automatic generation of test cases from SDL specifications," *SDL Newsletter*, no. 12, June 1988.

[3] Anne Bourguet-Rouger and Pierre Combes, "Exhaustive validation and test generation in elivis," in *SDL Forum'89: The Language at Work*, Ove Faergemand and Maria Manuela Marques, Eds. New York: North-Holland, 1989, pp. 231–245.

[4] Gang Luo and Junliang Chen, "Investigation & testing for SDL SAVE function," *J. Beijing Univ. of Posts and Telecommun.*, vol. 12, no. 4, pp. 34–40, Dec., 1989.

[5] Gang Luo, Anindya Das, and Gregor v. Bochmann, "Test selection based on SDL specification with save," in *SDL'91 Evolving Methods, Proc. 5th SDL Forum*, O. Faergemand and R. Reed, Eds. New York: North-Holland, 1991, pp. 313–324.

[6] F. Belina and D. Hogrefe, "The CCITT-specification and description language SDL," *Comput. Networks and ISDN Syst.*, vol. 16, pp. 311–341, 1989.

[7] R. Saracco and P. A. J. Tilanus, "CCITT SDL: Overview of language and its application," *Comput. Networks and ISDN Syst.*, vol. 13, no. 2, pp. 65–74, 1987.

[8] G. v. Bochmann *et al.*, "Fault models in testing," in *IFIP Trans., Protocol Testing Systems IV (Proc. IFIP TC6 4th Int. Workshop on Protocol Test Systems)*, Jan Kroon, Rudolf J. Heijink, and Ed Brinksma, Eds. New York: North-Holland, 1992, pp. 17–30.

[9] T. S. Chow, "Testing software design modeled by finite-state machines," *IEEE Trans. Software Eng.*, vol. 4, no. 3, pp. 178–187, 1978.

[10] S. Fujiwara, G. von Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi, "Test selection based on finite state models," *IEEE Trans. Software Eng.*, vol. 17, no. 6, pp. 591–603, June 1991.

[11] K. Sabnani and A. T. Dahbura, "A new technique for generating protocol tests", *ACM Comput. Commun. Rev.*, vol. 15, no. 4, pp. 36–43, 1985.

[12] S. Naito and M. Tsunoyama, "Fault detection for sequential machines by transition tours." in *Proc. FTCS*, 1981, pp. 238–243.

[13] T. Bolognesi and Ed Brinksma, "Introduction to the ISO specification language LOTOS," *Comput. Network and ISDN Syst.*, vol. 14, no. 1, pp. 25–59, 1987.

[14] S. Budkowski and P. Dembinski, "An introduction to Estelle: A specification language for distributed systems," *Comput. Network and ISDN Syst.*, vol. 14, no. 1, pp. 3–23, 1987.

[15] O. Faergemand and R. Reed, Eds., *SDL'91 Evolving Methods, Proc. 5th SDL Forum.* New York: North-Holland, 1991.

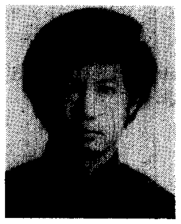
[16] B. Sarikaya, G. von Bochmann, and E. Cerny, "A test design methodology for protocol testing," *IEEE Trans. Software Eng.*, vol. SE-13, no. 9, pp. 989–999, Sept. 1987.

[17] F. Kristoffersen, L. Verhaard, and M. Zeeberg, "Test derivation for SDL based on ACTs," in *Proc. IFIP 5th Int. Conv. Formal Description Techniques*, M. Diaz and R. Groz, Eds., 1992.

[18] A. Kalnins, "Global state based automatic test generation for SDL," in *SDL'91 Evolving Methods, Proc. 5th SDL Forum*, O. Faergemand and R. Reed, Eds. New York: North-Holland, 1991, pp. 303–312.

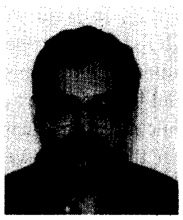
[19] D. Y. Lee and J. Y. Lee, "A well-defined Estelle specification for the automatic test generation," *IEEE Trans. Comput.*, vol. 40, no. 4, pp. 526–542, Apr. 1991.

[20] F. Belina, D. Hogrefe, and A. Sarma, *SDL with Applications from Protocol Specification.* Englewood Cliffs, NJ: Prentice-Hall, 1991.



**Gang Luo** (M'90) received the B.E. degree in computer and electrical engineering from Chongqing University, Chongqing, China, in 1982, and the M.E. degree in computer science and the Ph.D. degree in computer science and telecommunication from Beijing University of Posts and Telecommunications, Beijing, China, in 1987 and 1989, respectively.

He was an Assistant Engineer at No. 6 Research Institute of the China Ministry of Electronics Industry from 1982 to 1984. he served as a Lecturer and a leader of the Telecommunication Software research Group at Beijing University of Posts and Telecommunications from 1989 to 1990. He has been a Postdoctoral Fellow with the IDACOM–NSERC–CWARC Industrial Research Chair on Communication Protocols at the Université de Montréal, Montréal, P.Q., Canada, since 1990. His current research interests include telecommunication software engineering, software and protocol testing, formal specification techniques, software testability, software verification, telecommunication network protocols, and multimedia application.



**Anindya Das** received the M.Sc. degree in mathematics from the Indian Institute of Technology, Kanpur, India, in 1982, and the M.S. degree in computer science and engineering from the Indian Institute of Technology, Madras, India, in 1985, and the Ph.D. degree in electrical and computer engineering from Concordia University, Montréal, P.Q., Canada, in 1989.

He is currently an Assistant Professor in the Department of Computer Science and Operations Research, Université de Montréal, Montréal, P.Q., Canada, where his research includes communication protocol engineering, distributed systems management, distributed algorithms, and fault-tolerant computing.

**Gregor v. Bochmann**, for a photo and biography, please see page 42 of this issue of this TRANSACTIONS.