NEC
Empowered by Innovation

University of Waterloo

Software Testing and Quality Assurance

Theory and Practice

Chapter 10

Test Generation from FSM Models

Outline of the Chapter

- Characterizing Sequence
- Test Architectures
- Testing and Test Control Notation 3 (TTCN-3)
- Extended Finite-state Machines
- Test Generation from EFSM Models
- Additional Coverage Criteria for System Testing
- Summary

- State-oriented Model
- Points of Control and Observation
- Finite-state Machine (FSM)
- Test Generation from an FSM
- Transition Tour Method
- Testing with State Verification
- Unique Input/Output Sequence
- Distinguishing Sequence

State-oriented Model

- Software systems
  - State-oriented (Examples: Operating Systems)
  - Stateless (Example: compiler)
- State-oriented system: two parts
  - Control portion
  - Data portion

Figure 10.1: Spectrum of software systems

State-oriented Model

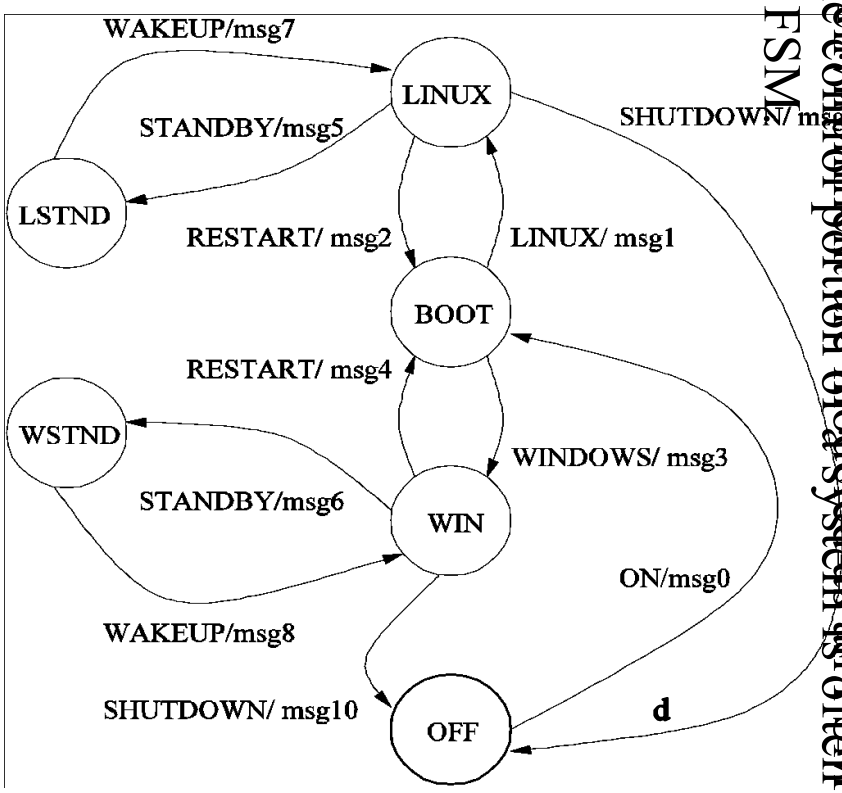
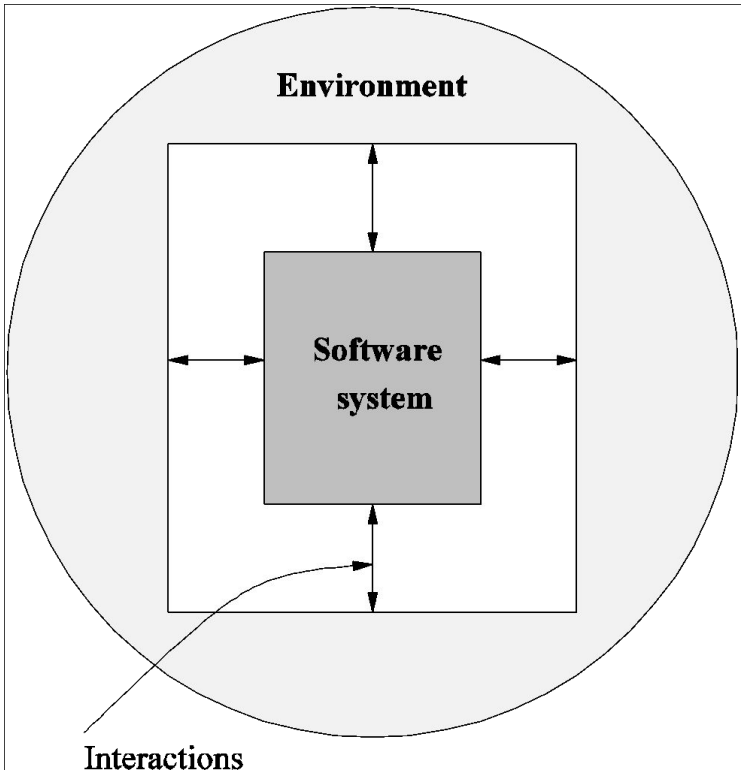


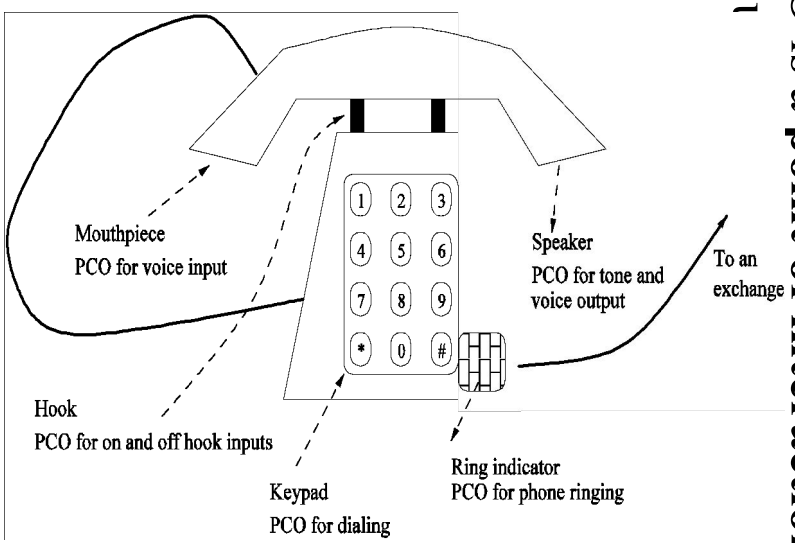
Figure 10.4: FSM model of a dual-boot laptop computer.

- Figure 10.5: The interactions between a system and its environment can be modeled as an FSM.



Points of Control and Observation

Figure 10.6: PCOs on a telephone

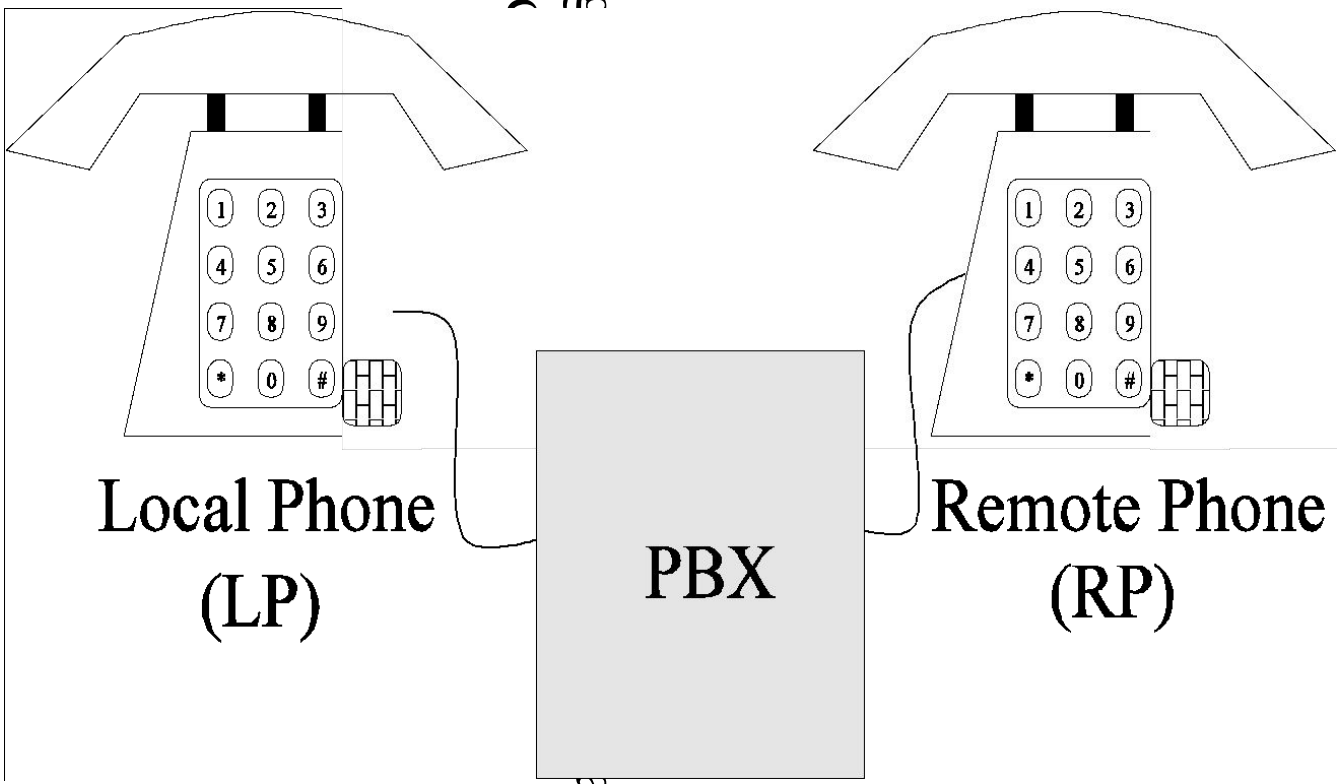


- A PCO is a **point of interaction** between a system and it

Table 10.1: PCOs for testing a telephone PBX.

PCO	IN/OUT
Hook	IN
Keypad	IN
Ringer	OUT
Speaker	OUT
Mouthpiece	IN

Points of Control and Observation



Finite-state Machine (FSM)

A finite-state machine (FSM) is a model of a private branch exchange (PBX).

Figure 10.8: FSM model of a PBX.

- $\langle S, I, O, s0, \delta, \lambda \rangle$, where
- S is a set of states.
- I is a set of inputs.
- O is a set of outputs.
- s0 is the initial state.
- δ: S x I $\rightarrow$ S (next state function)
- λ: S x I $\rightarrow$ O (output function)

Test Generation from an FSM

- Let M be the FSM model of a system and I_M be its implementation.
- An important testing task is to confirm if I_M behaves like M.
- Conformance testing: Ensure by means of testing that I_M conforms to its spec. M.
- A general procedure for conformance testing
  - Derive sequences of state-transitions from M.
    - Turn each state-transition into a test sequence.
    - Test I_M with a test sequence to observe whether or not I_M possesses the corresponding transition sequence.
  - The conformance of I_M with M can be verified by choosing enough state-transition sequences.

Transition Tour (TT) Method

- TT: It is a sequence of state-transitions from the initial state to the final state.
- Example: From Figure 10.8 Figure 10.10 Derived test case from the transition tour.

```
1     LP !OFH
2       START(TIMER1, d1)
3         LP ?DT                    PASS
4           CANCEL(TIMER1)
5             LP !ONH
6         LP ?OTHERWISE             FAIL
7           CANCEL(TIMER1)
8             LP !ONH
9         ?TIMER1                   FAIL
10          CANCEL(TIMER1)
11            LP !ONH
```

- Ideas in turning a TT into a test case.
  - The input/output in a test case are derived from a TT.
  - Unexpected inputs must be processed.
  - Indefinite waits must be avoided.

Figure 10.9: Interaction of a test sequence with an SUT.

Transition Tour (TT) Method

- **Coverage metrics** for FSM based testing
  - State coverage
    - Choose enough number of TTs to be able to cover **each state** at least once.
      - You can choose 3 TTs to cover **all** the states, but just 11 of the transitions.
  - Transition coverage
    - Choose enough number of TTs to be able to cover **each state-transition** at least once.

- State verification with
  - Unique Input/Output (UIO) sequences
  - Distinguishing sequences
  - Characterizing sequences

- Two functions associated with a state-transition
  - Output function: Easy to verify in the TT method.
  - Next-state function: Ignored in the TT method (drawback of the TT method).

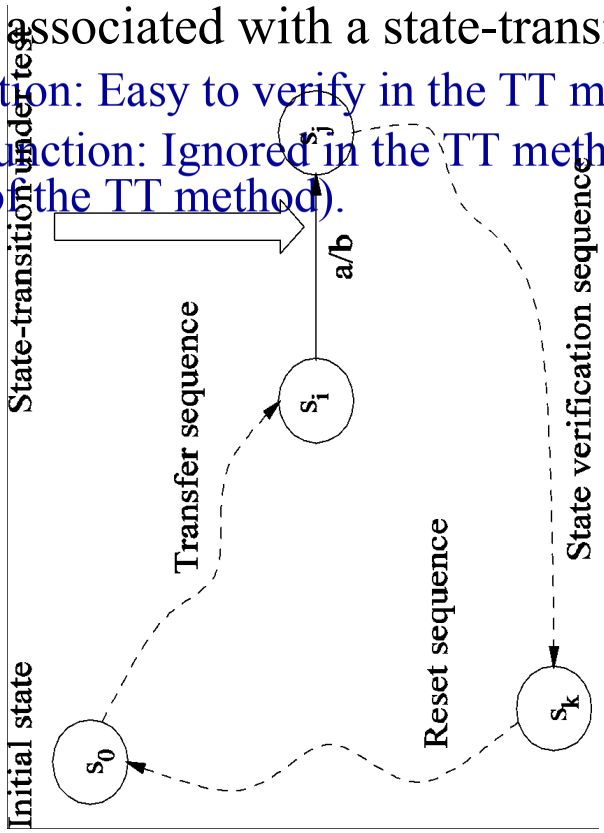


Figure 10.11: Conceptual model of a test case with state verification

Unique Input / Output Sequence

- Let X be an **input** sequence applied in a state s, and Y be the corresponding **output** sequence.
- X/Y is a **UIO sequence** for s if no other state produces output sequence Y in response to input X. Thus, X/Y is unique to s.
- Four assumptions about an FSM
  - Completely specified
  - Deterministic
  - Reduced
  - Strongly connected

Unique Input / Output Sequence

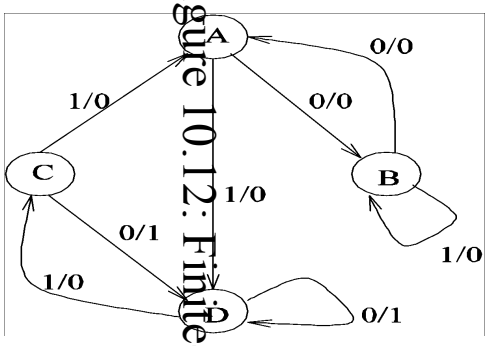


Figure 10.12: Finite-state m

UIO sequences		
State	Input sequence	Output sequence
A	010	000
B	010	001
C	1010	0000
D	11010	00000

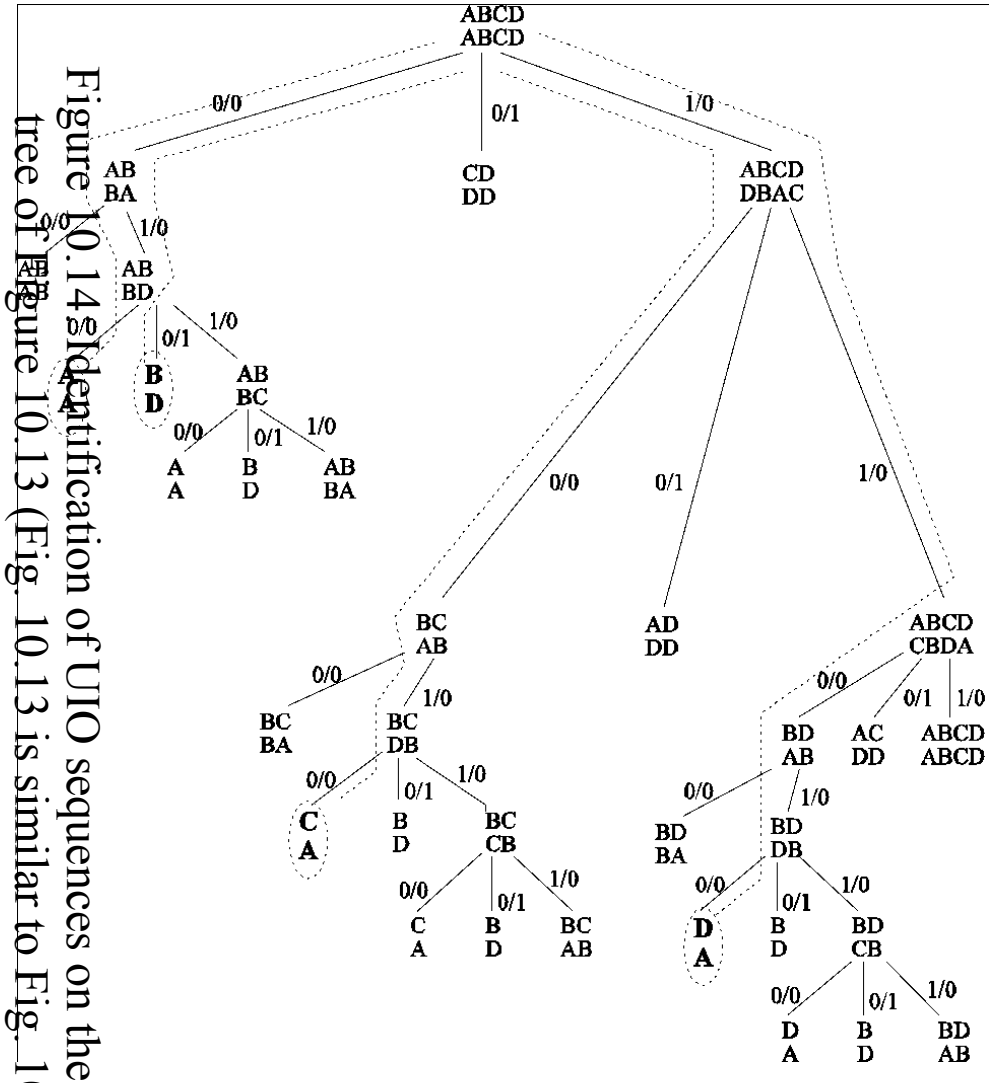


Figure 10.14 Identification of UIO sequences on the UIO tree of Figure 10.13 (Fig. 10.13 is similar to Fig. 10.14).

Distinguishing Sequence

- Let X be an **input** sequence.
- X is a **distinguishing sequence** for an **FSM,** if each state produces a **unique** (i.e. different) output sequence in response to X.
- Four assumptions about an FSM
  - Completely specified
  - Deterministic
  - Reduced
  - Strongly connected

Distinguishing Sequence

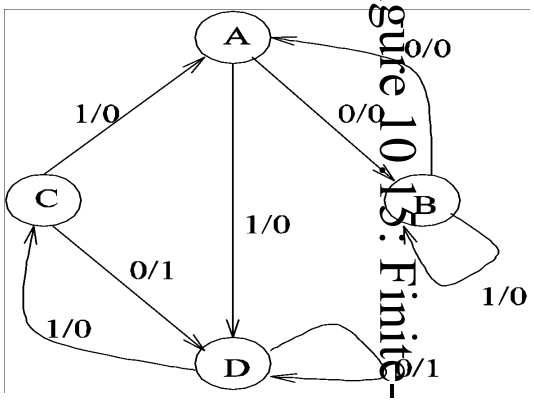


Figure 10.15: Finite-state machine G2.

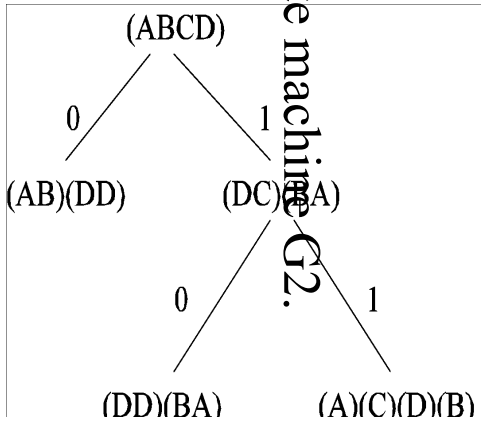


Table 10.9: Ou
sequence 11 in different states.

esponse to input

Present state	Output sequence
A	00
B	11
C	10
D	01

Characterizing Sequence

- Some FSMs do not have a D-sequence.
  - Use characterizing sets.
- Characterizing Sets (CS)
  - A CS for a state s i
  - each sequence is ap
  - sequences is unique
  - Thus s is uniquely i

such that, when
set of output

Figure 10.17: An FSM that does not possess a distinguishing

Figure 10.18: DS-tree for FSM (Figure 10.17).

- State verification is still possible.

Characterizing Sequence

Starting States	Output Generated by $W_1 = aba$
A	xyx
B	yyy
C	yyx
D	xyx

Outpu
ure 10.

Starting States	Output Generated by $W_2 = ba$
A	yx
B	yx
C	yy
D	yy

ed by the
W2.

Figure 10.10: Output sequences generated by the FSM of Figure 10.17 as a response to W1.

Characterizing Sequence

Test Sequence Table				
Step	**Current State**	**Next State**	**Message to SUT**	**Message from SUT**
Apply $T(D)$:				
1	A	D	b	y
Test the Transition (D, A, a/x):				
2	D	A	a	x
Apply W_1 :				
3	A	A	a	x
4	A	D	b	y
5	D	A	a	x
Apply RI :				
6	A	D	b	y
7	D	A	a	x
8	A	D	b	y
9	D	A	a	x
10	A	D	b	y
11	D	A	a	x
Apply $T(D)$:				
12	A	D	b	y
Test the Transition (D, A, a/x):				
13	D	A	a	x
Apply W_2 :				
14	A	D	b	y
15	D	A	a	x
Apply RI :				
16	A	D	b	y
17	D	A	a	x
18	A	D	b	y
19	D	A	a	x
20	A	D	b	y
21	D	A	a	x

ition (D,

Test Architectures

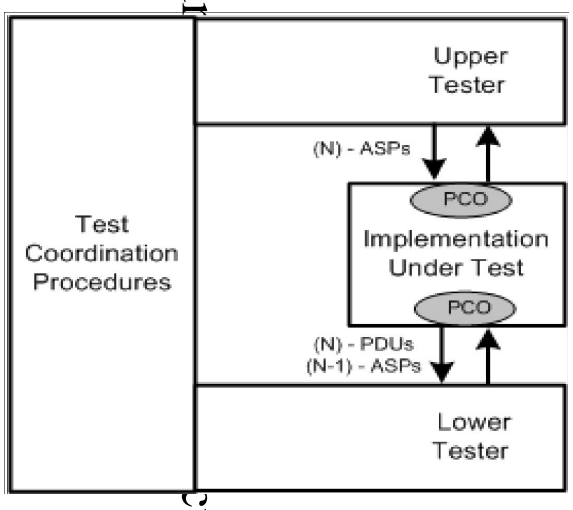
- A test architecture is a certain configuration of
  - an **Implementation Under Test** (IUT)
  - one or more **test entities**,
  - one or two **PCOs**, and
  - a communication service provide.
- Common test architectures
  - Local Architecture
  - Distributed Architecture
  - Coordinated Architecture
  - Remote Architecture

Figure 1(ntity in the OSI refere

Test Architectures

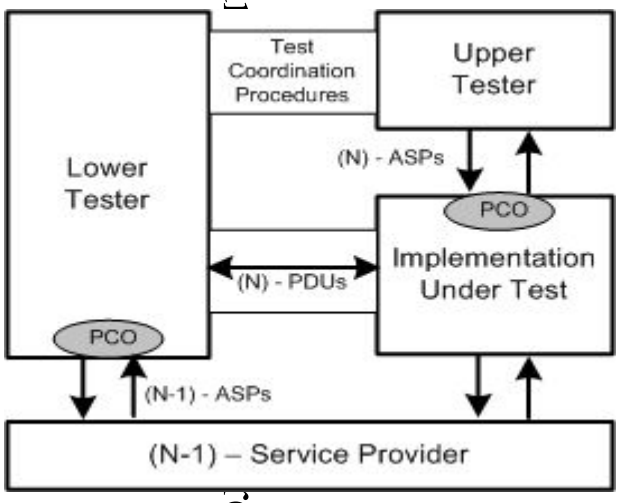
- Local Architecture

Figure. Local Architecture.



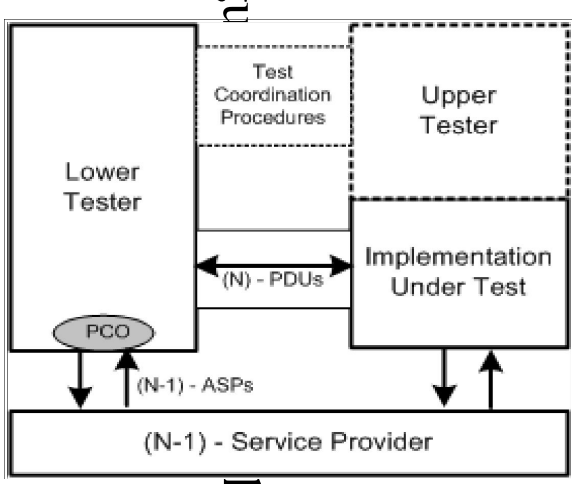
- Distributed Architecture

Figure. Distributed Architecture.



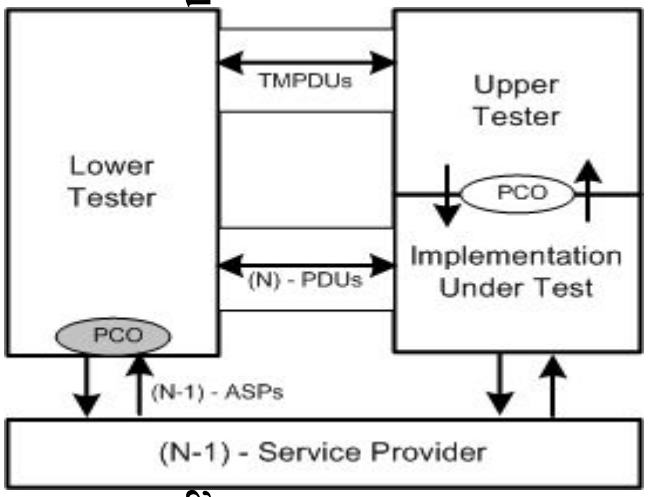
Test Architectures

- Remote Architecture



Figu hitecture.

- Coordinated Architecture



Figu architecture.

- TTCN-3
  - A language for specifying test cases.
  - Predecessors were TTCN-1 and TTCN-2 (Tree and Tabular Combined Notation)
  - Standardized by ETSI (European Telecom. Standards Institute)
- Core features of TTCN-3
  - Module
  - Data types
  - Templates
  - Ports
  - Components
  - Test Cases

Testing and Test Control Notation 3 (TTCN-3)

• Module

```
/* One can document a module by writing comments in this way. */
// Additional comments can be included here.
module ExampleTestModule1 { // A module can be empty.
      // First, define some data to be used in the control part
      const integer MaxCount := 15;
      constant integer UnitPacket = 256;
      // More data can be defined here ...

      // Second, specify the control part to execute
     control { // The control part is optional
           var integer counter := 0;
           var integer loopcount := MaxCount;
           const integer PacketSize := UnitPacket * 4;

           // Specify more execution behavior here ...

     } // End of the control part
} // end of module TestCase1
```

in

Testing and Test Control Notation 3 (TTCN-3)

- Types, subtypes, and messages

```
type integer TCPPort ( 0 .. 65535 ); // a 16 bit unsigned number
type charstring IPUserProtocol ( "TCP", "UDP", "OSPF", "RIP" );

template MyMessage SFCRequest( Identification id, Input Ival) := {
     identification := id,
     msgtype        := Request,
     input          := Ival,
     response       := omit // "omit" is a keyword
}

template MyMessage SFCResponse( Identification id, Response Rval) := {
     identification := id,
     msgtype        := Response,
     input          := ?, // This means the field can contain any value
     response       := Rval
}
```

Testing and Test Control Notation 3 (TTCN-3)

- Ports

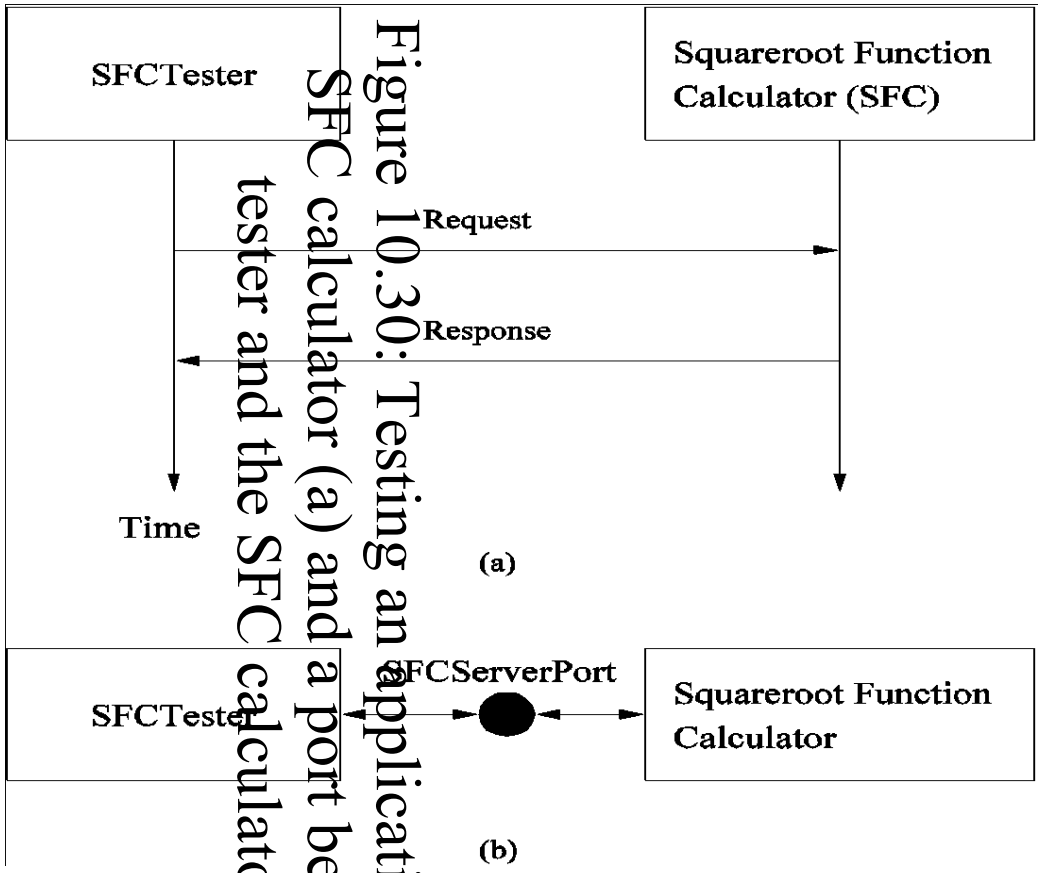


Figure 10.30: Testing an application called SFC calculator (a) and a port between the tester and the SFC calculator (b).

Testing and Test Control Notation 3 (TTCN-3)

- Ports and components

```
type port SFCPort message { // The SFCPort type has a "message" semantics
      inout MyMessage       // The SFCPort type is of inout type handling
                            // messages of type MyMessage
}
```

F

: Associat

compon

rt with a

```
type component SFCTester {
      port SFCPort SFCServerPort
}
```

31: Defining a port type.

Testing and Test Control Notation 3 (TTCN-3)

Figure f

```
// A test case description with alternative behavior
testcase SFCtestcase1() runs on SFCTester {
      timer responseTimer; // Define a timer
      SFCPort.send(SFCRequest(7, 625));
      responseTimer.start(5.0);

      alt { // Now handle three alternative cases ...
            // Case 1: The expected result of computation is received.
            [] SFCPort.receive(SFCResponse(7, 25)) {
                  setverdict(pass);
                  responseTimer.stop;
               }

            // Case 2: An unexpected result of computation is received.
            [] SFCPort.receive {
                  setverdict(fail);
                  responseTimer.stop;
               }
            // Case 3: No result is received within a reasonable time.
            [] responseTimer.timeout {
                  setverdict(fail);
               }
      }
      stop;
} // End of test case
```

Testing and Test Control Notation 3 (TTCN-3)

- Test case execution

```
module ExampleTestModule2 {
      // Define variables and constants to be used ...
      // Define templates to be used ...
      // Define ports to be used ...
      // Associate test components with ports ...
      // Define test cases, such as SFCtestcase1 ...

      control {
            execute( SFCtestcase1() );
      }
}
```

case.

Extended Finite-state Machines

- Two conceptual components of a software system are
  - Flow of control
  - Manipulation of data
    - Manipulate local variables.
    - Start and stop timers.
    - Create instances of processes.
    - Compare values and make control-flow decisions.
    - Access databases.
- There is a need for modeling a software system as an EFSM.
- We consider the Specification and Description Languages (SDL).
  - The basic concepts in SDL are as follows:
    - System
    - Behavior
    - Data
    - Communication

Extended Finite-state Machines

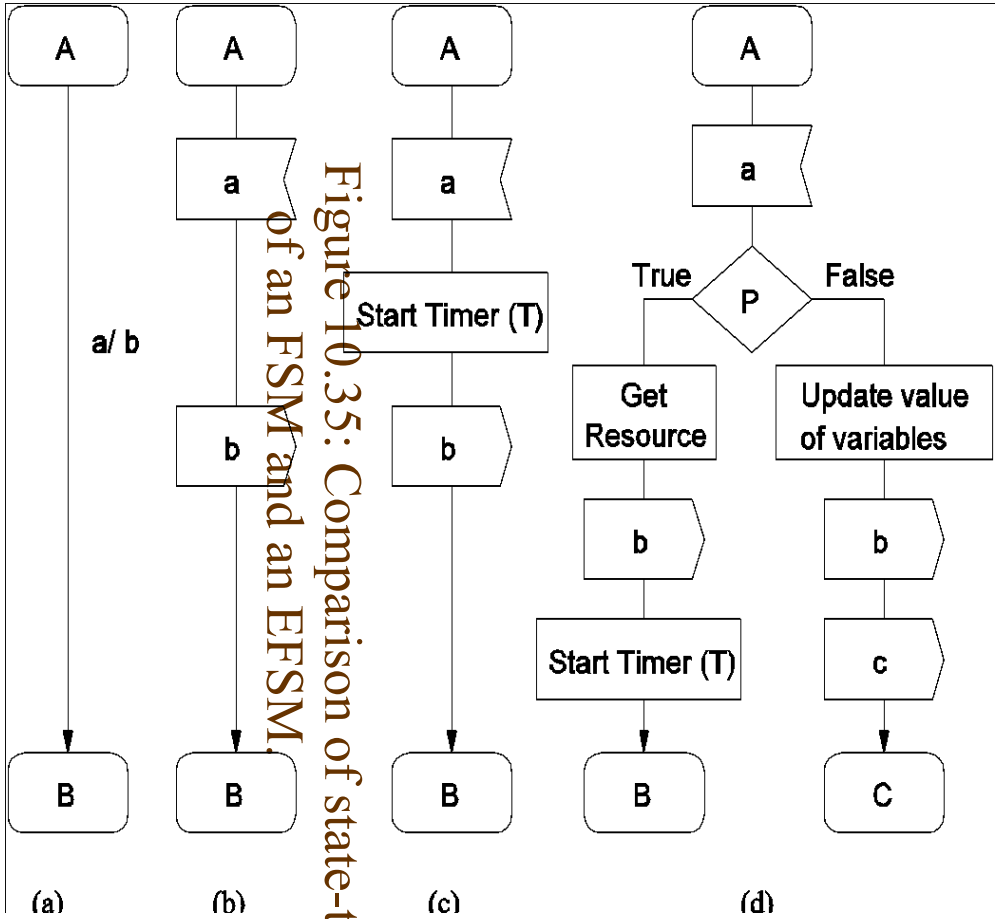


Figure 10.35: Comparison of state-transitions of an FSM and an EFSM.

- Let E be an EFSM and P_E be a program implementing E.
- Goal: Test that P_E behaves as E.
- Basic idea
  - **Phase 1**: Identify a set of state-transition sequences such that each sequence of state-transitions represents a common use sequence.
  - **Phase 2**: Design a test case from each state-transition sequence.
- Phase 1
  - Pay attention to the following.
    - Perform tests to ensure that the **system under test** (SUT) produces expected sequences of outcomes in response to input sequences.
    - Perform tests to ensure that the system under test takes the right actions when a timeout occurs.
    - Perform tests to ensure that the system under test has appropriately implemented other task blocks, such as resource allocation, database accesses, etc.
  - Coverage criteria: state coverage and transition coverage.

- Phase 2
  - Transform the **inputs** and **outputs** in a transition tour into **outputs** and **inputs**, respectively.
  - Augment the above core test behavior with **"otherwise"** events to be able to handle exception events.
  - Augment the above test behavior with **timers.**

Test Generation from EFSM Models

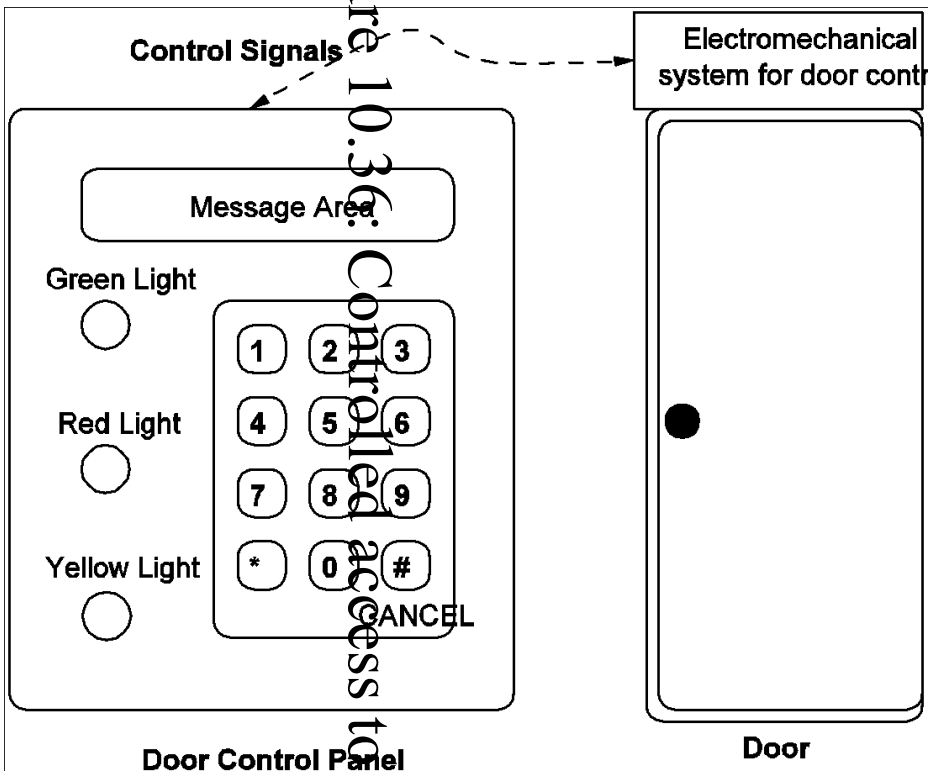


Figure 10.36: Controlled access to a door.

Test Generation from EFSM Models

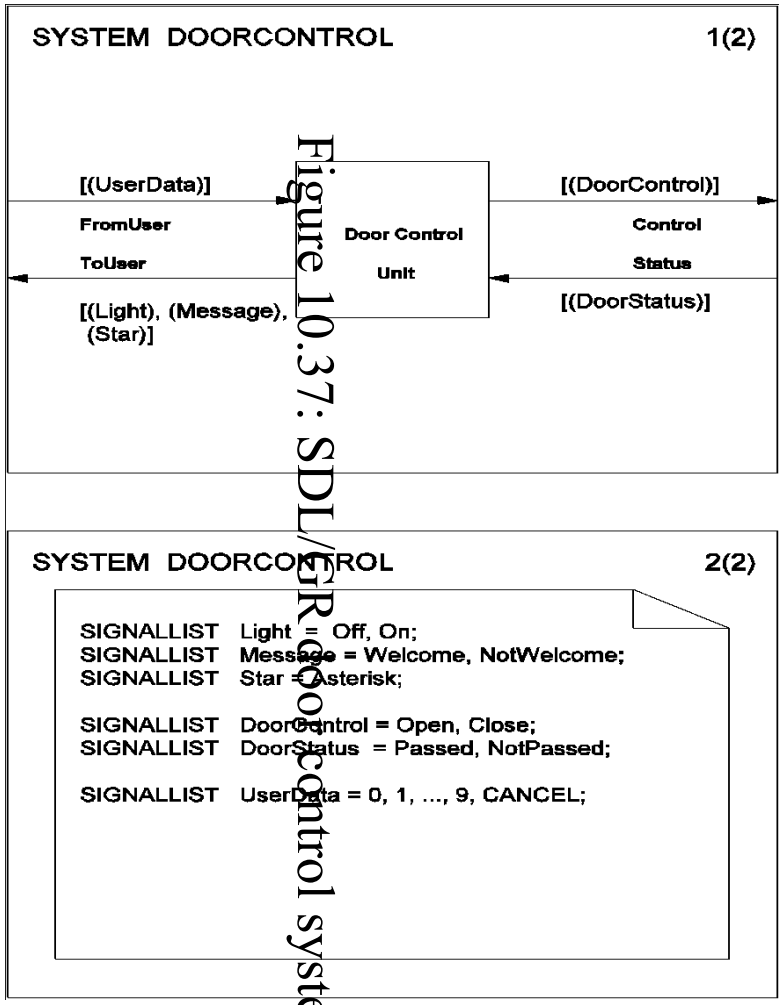
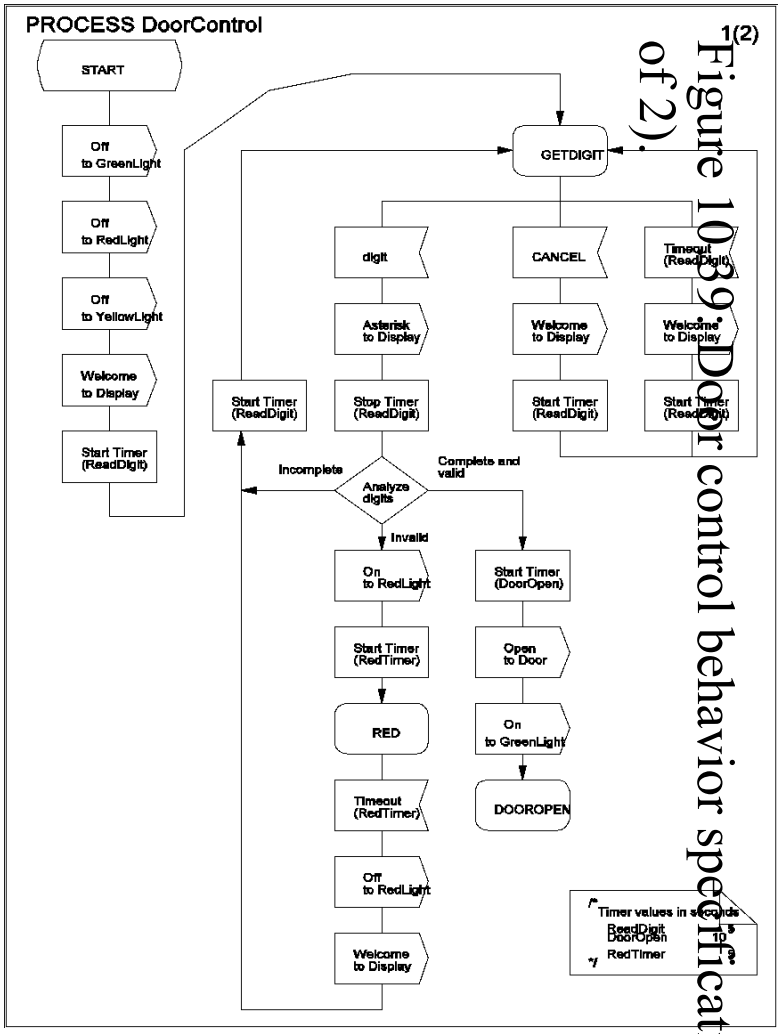
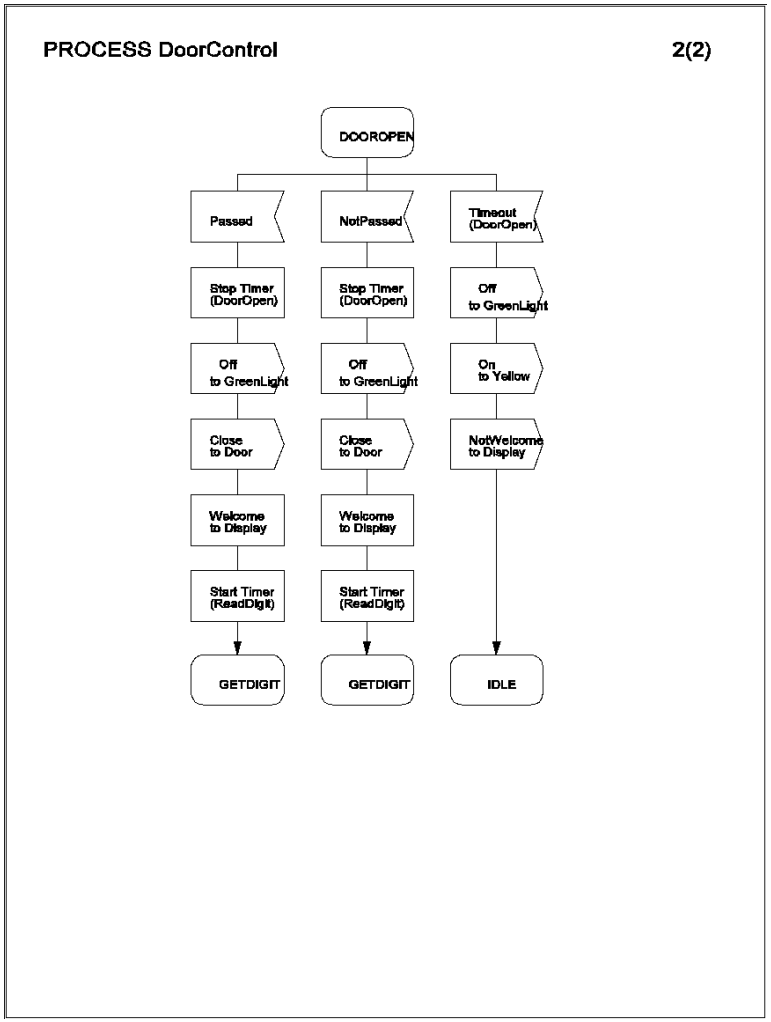


Figure 10.37: SDL/GR door control system.

Test Generation from EFSM Models





- Figure 10.39: Door control behavior specification (2 of 2).

Test Generation from EFSM Models

- Figure 10.41: Testing the door control system.

GETDIGIT □ … GETDIGIT □ … GETDIGIT □ … DOOROPEN □ GETDIGIT.

Figure 10.40: A transition tour from the door control system of Figs. 10.38 and 10.39

Test Generation from EFSM Models


```
1.  label label1 KEYPAD.send(digit);
2.    DISPLAY.receive(Asterisk);
3.     if (NOT enough number of digits) goto label1; // Not in TTCN-3 form yet.
4.        DOOR.receive(Open);
5.           GREENLIGHT.receive(On);
6.              DOOR.send(Passed);
7.                 GREENLIGHT.receive(Off);
8.                    DOOR.receive(Close);
9.                       DISPLAY.receive(Welcome);
10.                         setverdict(Pass);
```

r obtained

Test Generation from EFSM Models


```
1.   count := 0; // Count is of type integer.
2.   label label1 KEYPAD.send( digit );
3.       DISPLAY.receive( Asterisk );
4.         count := count + 1;
5.           if (count < 4) goto label1;
6.           else {
7.               DOOR.receive( Open );
8.                  GREENLIGHT.receive( On );
9.                     DOOR.send( Passed );
10.                       GREENLIGHT.receive( Off );
11.                          DOOR.receive (Close);
12.                             DISPLAY.receive(Welcome);
13.                                setverdict(Pass);
14.          };
```

refining

Test Generation from EFSM Models


```
1.  count := 0; // Count is of type integer.
2.  label label1 KEYPAD.send( digit );
3.  alt {
4.      [] DISPLAY.receive( Asterisk );
5.         count := count + 1;
6.         if (count < 4) goto label1;
7.         else {
8.                alt {
9.                   [] DOOR.receive( Open );
10.                    alt {
11.                         [] GREENLIGHT.receive( On );
12.                            DOOR.send( Passed );
13.                            alt {
14.                                  [] GREENLIGHT.receive( Off );
15.                                    alt {
16.                                         [] DOOR.receive (Close);
17.                                            alt {
18.                                                 [] DISPLAY.receive(Welcome);
19.                                                    setverdict(Pass);
20.                                                 [] DISPLAY.receive(?);
21.                                                    setverdict(Fail);
22.                                             }
23.                                         [] DOOR.receive(?);
24.                                            setverdict(Fail);
25.                                    }
26.                                 [] GREENLIGHT.receive(?);
27.                                    setverdict(Fail);
28.                            }
29.                         [] GREENLIGHT.receive(?);
30.                            setverdict(Fail);
31.                   }
32.                [] DOOR.receive(?);
33.                   setverdict(Fail);
34.             }
35.          } // end of else
36.     [] DISPLAY.receive(?);
37.        setverdict(Fail);
38. }
```

Fig. 10.16

Test Generation from EFSM Models


```
1.  count := 0; // Count is of type integer.
2.  label label1 KEYPAD.send( digit );
3.  Timer1.start(d1);
4.  alt {
5.      [] DISPLAY.receive( Asterisk );
6.         Timer1.stop;
7.         count := count + 1;
8.         if (count < 4) goto label1;
9.         else {
10.               Timer2.start(d2);
11.               alt {
12.                  [] DOOR.receive( Open );
13.                     Timer2.stop; Timer3.start(d3);
14.                     alt {
15.                          [] GREENLIGHT.receive( On );
16.                             Timer3.stop;
17.                             DOOR.send( Passed );
18.                             Timer4.start( d4 );
19.                             alt {
20.                                  [] GREENLIGHT.receive( Off );
21.                                     Timer4.stop; Timer5.start( d5 );
22.                                     alt {
23.                                          [] DOOR.receive (Close);
24.                                             Timer5.stop; Timer6.start( d6 );
25.                                             alt {
26.                                                  [] DISPLAY.receive(Welcome);
27.                                                     Timer6.stop; setverdict(Pass);
28.                                                  [] DISPLAY.receive(?);
29.                                                     Timer6.stop; setverdict(Fail);
30.                                                  [] Timer6.timeout;
31.                                                     setverdict(Inconc);
32.                                              }
33.                                          [] DOOR.receive(?);
34.                                             Timer5.stop; setverdict(Fail);
35.                                          [] Timer5.timeout;
36.                                             setverdict(Inconc);
37.                                     }
38.                                  [] GREENLIGHT.receive(?);
39.                                     Timer4.stop; setverdict(Fail);
40.                                  [] Timer4.timeout;
41.                                     setverdict(Inconc);
42.                             }
43.                          [] GREENLIGHT.receive(?);
44.                             Timer3.stop; setverdict(Fail);
45.                          [] Timer3.timeout;
46.                             setverdict(Inconc);
47.                     }
48.                  [] DOOR.receive(?);
49.                     Timer2.stop; setverdict(Fail);
50.                  [] Timer2.timeout;
51.                     setverdict(Inconc);
52.               }
53.         }
```

Additional Coverage Criteria for System Testing

- PCO coverage
  - Select test cases such that the SUT receives an event at each input PCO and produces an event at each output PCO.
- Sequence of events at PCOs
  - Select test cases such that common sequences of inputs and outputs occur at the PCOs.
- Events occurring in different contexts
  - An event generated at a PCO may have different meanings at different times.
- Inopportune events
  - Inopportune events are normal events which occur at an inappropriate time.

Summary

- TTCN-3
  - Data types
  - Modules
  - Ports
  - Templates
- Testing EFSM based systems
  - Identify transition sequences
  - Turn each sequence into a test case
    - Input/output □ Output/input
    - "Otherwise" events
    - Timers
  - Coverage metrics
    - PCO coverage
    - Sequences of events at each PCO
    - Events occurring in different contexts
    - Inopportune events
- Software systems
  - Stateless
  - State-oriented
    - Control portion is modeled by an FSM or an EFSM.
- Testing FSM based systems
  - Transition tour method
  - State verification method
- State verification techniques
  - Distinguishing sequence
  - UIO sequence
  - Characterizing sequence
- Coverage metrics
  - State coverage
  - State-transition coverage
- Test architectures
  - Local
  - Distributes