Test Generation based on Finite State Models

W. Eric Wong
Department of Computer Science
The University of Texas at Dallas
ewong@utdallas.edu
http://www.utdallas.edu/~ewong

Speaker Biographical Sketch

- Professor & Director of International Outreach
  Department of Computer Science
  University of Texas at Dallas
- Guest Researcher
  Computer Security Division
  National Institute of Standards and Technology (NIST)
- Vice President, IEEE Reliability Society
- Secretary, ACM SIGAPP (Special Interest Group on Applied Computing)
- Principal Investigator, NSF TUES (Transforming Undergraduate Education in Science, Technology, Engineering and Mathematics) Project
  - *Incorporating Software Testing into Multiple Computer Science and Software Engineering Undergraduate Courses*
- Founder & Steering Committee co-Chair for the SERE conference (*IEEE International Conference on Software Security and Reliability*) (http://paris.utdallas.edu/sere13)

Learning Objectives

- What are Finite State Machines (FSM)?
- The W method for test generation
- The Wp method for test generation
- Automata theoretic versus control-flow based test generation
- What are Extended Finite State Machines (EFSM)?
- What are Communicating Extended State Machines (CEFSM)?
  - Architectural design in SDL
- EFSM-based test generation
- CEFSM-based test generation

Where Are These Methods Used?

- Conformance testing of communications protocols
  - Finite state machines are widely used in modeling of different kinds of systems.
  - Testing of any system/subsystem modeled as a finite state machine, e.g. elevator designs, automobile components (locks, transmission, stepper motors, etc.), steam boiler control, etc.
  - Generation of tests from FSM assists in testing the conformance of implementations to the corresponding FSM model
- White box-based coverage testing for SDL design specifications, EFSMs, CFEMSs, reachability graphs, etc.

What is an Finite State Machine?

- A finite state machine, abbreviated as FSM, is an abstract representation of behavior exhibited by some systems.
- An FSM is derived from application requirements. For example, a network protocol could be modeled using an FSM.
- Not all aspects of an application's requirements are specified by an FSM. For example, real time requirements and performance requirements cannot be specified by an FSM.

Requirements Specification or Design Specification?

- An FSM could serve as a *specification* of the required behavior or as a *design* artifact according to which an application is to be implemented.
  - The role assigned to an FSM depends on whether it is a part of the *requirements specification* or of the *design specification*.
  - FSMs are part of UML 2.0 design notation.

Finite State Machines with Output

- Mealy Machine (due to G. H. Mealy -1955 publication)
  - Outputs corresponds transitions between states.
- Moore Machine (due to E. F. Moore -1956 publication)
  - Outputs are determined only by the states

Mealy Machine (1)

- A Mealy machine M = {S, I, O, f, g, s_o}
  - S: a finite set of states
  - S_o: the start state (a.k.a. initial state) contained in S
  - I: a finite input alphabet
  - O: a finite output alphabet
  - f: **a transition function** that maps a state/input pair to the next state (S x I → S)
  - g: **an output function** that maps a state/input pair to an output (S x I → O)

State Diagram Representation of a Mealy FSM

- A state diagram is a *directed graph* where each node is a state and each edge is a transition between states
- Each transition from a state can be triggered by an input and produce an output
  - Input x Current State → Output x Next State

Tabular Representation of a Mealy FSM

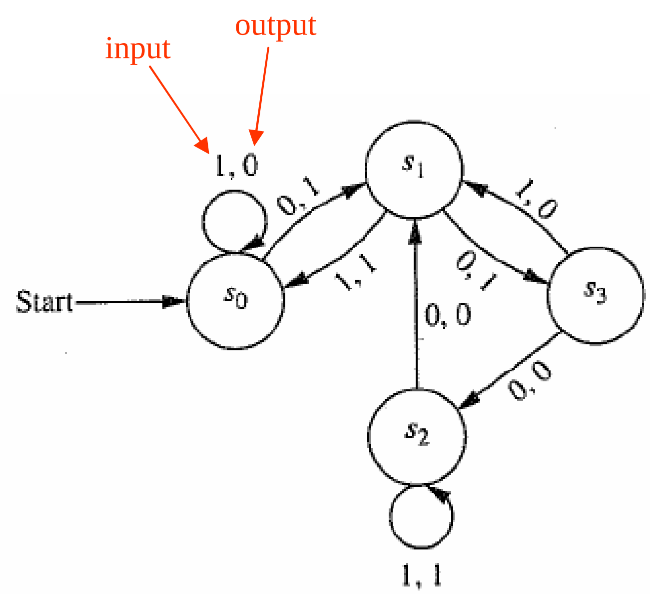
- A table is often used as an alternative to the state diagram
- The table consists of two sub-tables that consist of one or more columns each.
  - The left sub-table is the *input* sub-table.
  - The rows are labeled by the states of the FSM.
  - The right sub-table is the *output* sub-table.

Mealy Machine (2)

- An example of a Mealy machine

TABLE

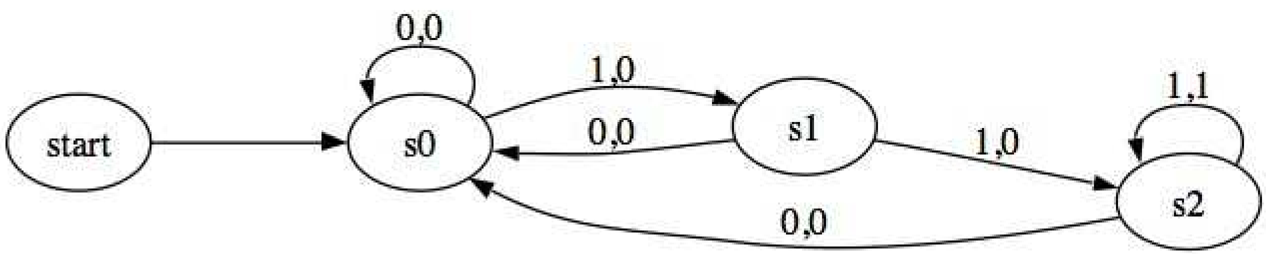
State	*f* Input 0	*f* Input 1	*g* Input 0	*g* Input 1
s_0	s_1	s_0	1	0
s_1	s_3	s_0	1	1
s_2	s_1	s_2	0	1
s_3	s_2	s_1	0	0



The state diagram for the FSM shown in the table

Mealy Machine (3)

- A Mealy machine that outputs 1 if and only if the input string read so far ends with 111. This machine is a *language recognizer.*



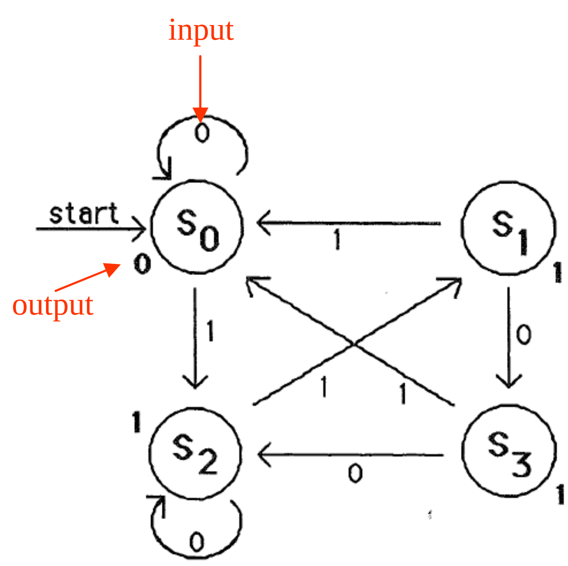
Moore Machine (1)

- A Moore machine $M = \{S, I, O, f, g, s_o\}$
  - S: a finite set of states
  - S_o: the start state (a.k.a. initial state) contained in S
  - I: a finite input alphabet
  - O: a finite output alphabet
  - f: a transition function that maps a (current) state/input pair to the next state $(S \times I \rightarrow S)$
  - g: an output function that maps a state to an output $(S \rightarrow O)$

Moore Machine (2)

- In a Moore machine, the output at each transition is only dependent on the *state*, rather than a state/input pair
- In other words, *the output after transition is determined solely by the final state of the transition*.

State	*f* Input 0	*f* Input 1	*g*
s_0	s_0	s_2	0
s_1	s_3	s_0	1
s_2	s_2	s_1	1
s_3	s_2	s_0	1



Deterministic versus Nondeterministic

- A deterministic finite automaton: For each pair of state and input value, there is a unique next state given by the transition function.
- A nondeterministic finite automaton: There may be *several possible* next states for each pair of state and input value.
  - If the language L is recognized by a nondeterministic finite state automaton $\mathcal{M}_1$, then L is also recognized by a deterministic finite state automaton $\mathcal{M}_2$.

Properties of FSM

- Completely specified: An FSM $\mathcal{M}$ is said to be completely specified if from each state in $\mathcal{M}$ there exists a transition for each input symbol.
- Strongly connected: An FSM $\mathcal{M}$ is considered strongly connected if for each pair of states $(s_i\ s_j)$ there exists an input sequence that takes $\mathcal{M}$ from state s_i to s_j.

V-Equivalence of Two States

- V-equivalence: Let $\mathcal{M}_1=(I, O, S_1, s^1{}_0, f_1, g_1)$ and $\mathcal{M}_2=(I, O, S_2, s^2{}_0, f_2, g_2)$ be two FSMs (Mealy Machines). Let V denote a set of non-empty strings over the input alphabet I, that is, $V \subseteq I^+$.

- Let s_i and s_j, $i \neq j$, be two states of machines $\mathcal{M}_1$ and $\mathcal{M}_2$, respectively. s_i and s_j are considered *V-equivalent* if $g_1(s_i, v)=g_2(s_j, v)$ for all v in V.

- Stated differently, states s_i and s_j are considered *V-equivalent* if $\mathcal{M}_1$ and $\mathcal{M}_2$, when excited in states s_i and s_j, respectively, yield *identical output sequences.*

Equivalence of Two States

- States s_i and s_j are said to be *equivalent* if $g_1(s_i, v)=g_2(s_j, v)$ for any set V
  - If s_i and s_j are not equivalent then they are said to be *distinguishable*
- We write $s_i{=}s_j$ if states s_i and s_j are equivalent, and $s_i \neq s_j$ when they are distinguishable

Equivalence of Two FSMs

- Machines $\mathcal{M}_1$ and $\mathcal{M}_2$ are said to be ***equivalent*** if
  - For each state α in $\mathcal{M}_1$ there exists a state α' in $\mathcal{M}_2$ such that α and α' are equivalent
  - For each state β in $\mathcal{M}_2$ there exists a state β' in $\mathcal{M}_1$ such that β and β' are equivalent
- Machines that are not equivalent are considered ***distinguishable***.
- If $\mathcal{M}_1$ and $\mathcal{M}_2$ are strongly connected, then they are equivalent if their respective initial states, s^1_0 and s^2_0, are equivalent
- We write $\mathcal{M}_1=\mathcal{M}_2$ if machines $\mathcal{M}_1$ and $\mathcal{M}_2$ are equivalent, and $\mathcal{M}_1\neq\mathcal{M}_2$ when they are distinguishable

K-Equivalence

- States $s_i \in S_1$ and $s_j \in S_2$ are considered *k-equivalent* if when excited by any input of length *k*, yield identical output sequences
  - States that are not k-equivalent are considered *k-distinguishable*
- It is also easy to see that if two states are *k-distinguishable* for any k>0 then they are also distinguishable for any $n \geq k$

Minimal Machine

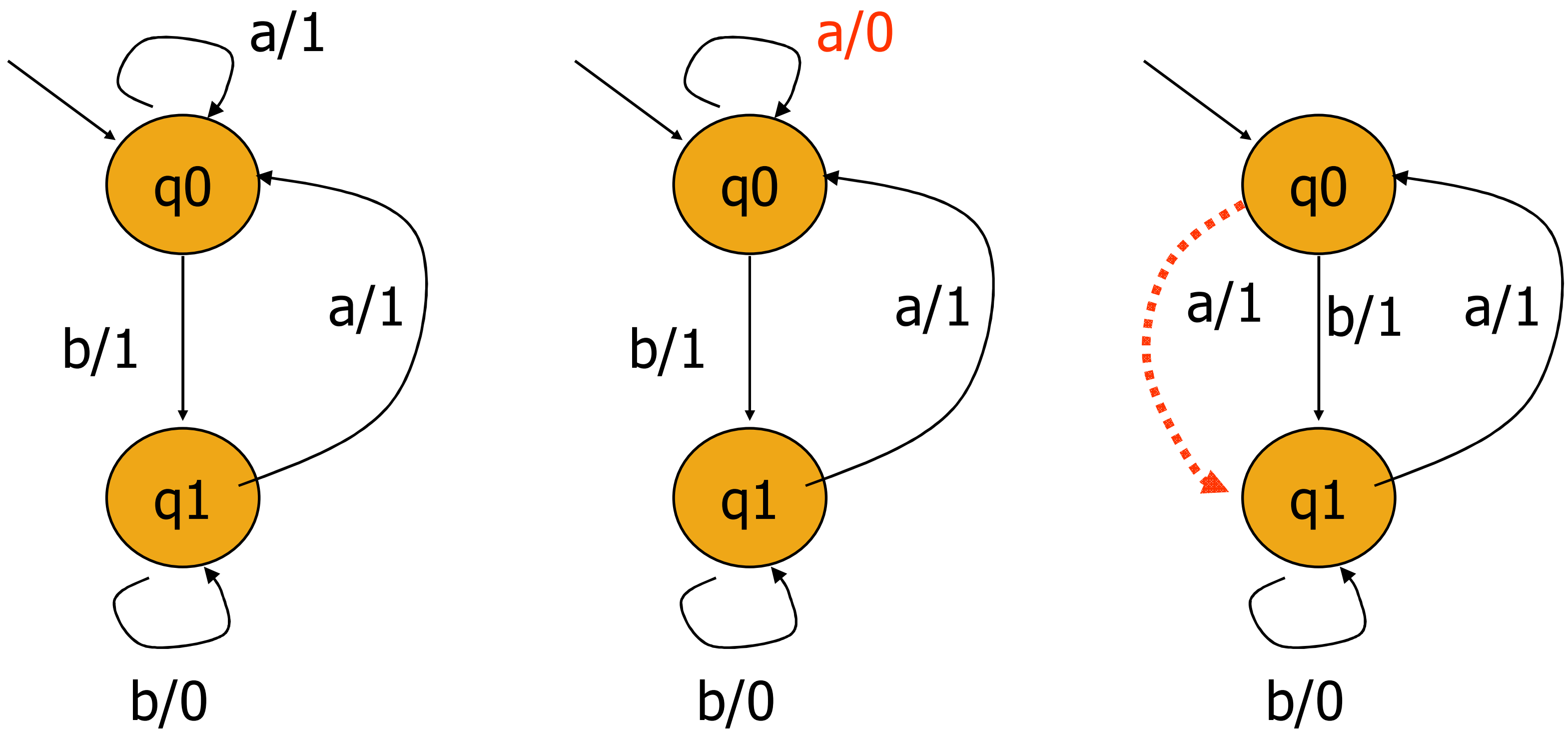
- An FSM $\mathcal{M}$ is considered *minimal* if the number of states in $\mathcal{M}$ is less than or equal to any other FSM equivalent to $\mathcal{M}$.

Faults in Implementation

- An FSM serves to specify the correct *requirement* or *design* of an application. Hence tests generated from an FSM target faults related to the FSM itself.

- *What faults are targeted by the tests generated using an FSM?*

Fault Model (1)

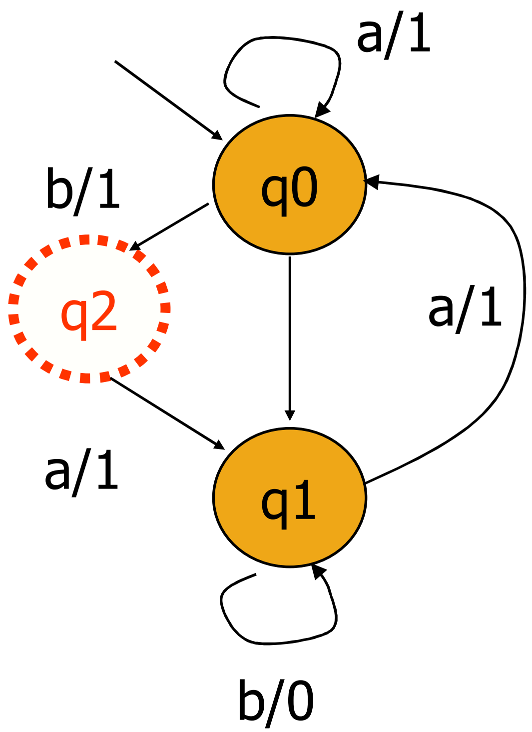


Correct design

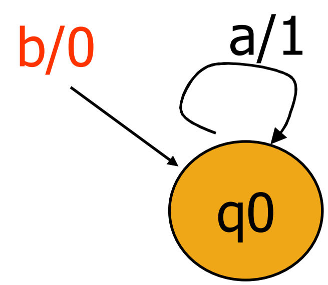
Operation error

Transfer error

Fault Model (2)



Extra state error



Missing state error

Test Generation using the W Method

Assumption

- $\mathcal{M}$ is ***completely*** specified, ***minimal***, ***connected***, and ***deterministic***
- $\mathcal{M}$ starts in ***a fixed initial state***
- $\mathcal{M}$ can be ***reset accurately to the initial state***. A *null* output is generated during the reset operation
- $\mathcal{M}$ and the implementation under test have the ***same input alphabet***

Steps of the W Method

- Step 1: Estimate the maximum number of states (*m*) in the correct implementation of the given FSM $\mathcal{M}$
- Step 2: Construct the characterization set W for $\mathcal{M}$
- Step 3: (1) Construct the *testing tree* for $\mathcal{M}$ and
  (2) Generate the transition cover set P from the testing tree
- Step 4: Construct set Z from W and *m*
- Step 5: Test generation and execution

Step 1: Estimation of m

Estimation of *m*

- This is based on a knowledge of the implementation. In the absence of any such knowledge, let $m = |S|$ (number of states in the given FSM)

Step 2: Construction of W

Step 2.1: Construction of the *K*-Equivalence Partitions

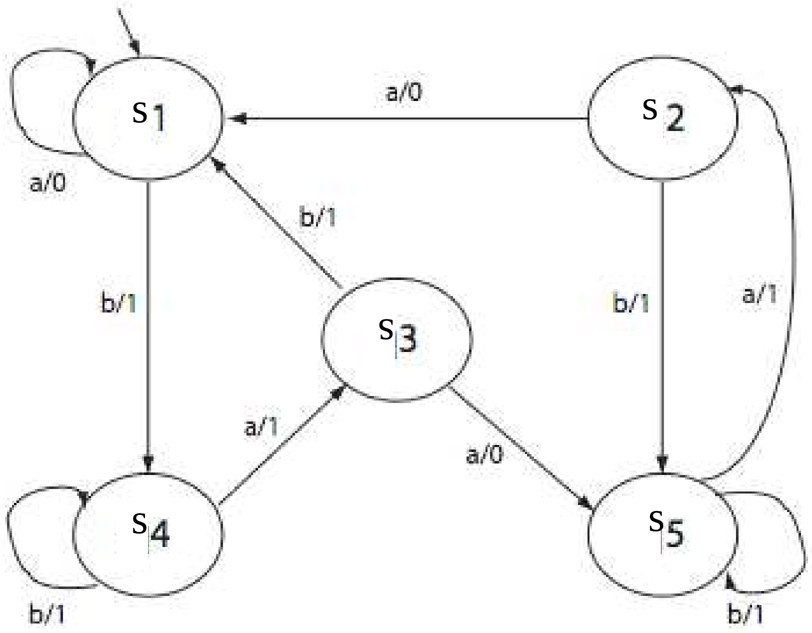
Step 2.2: Derivation of the Characterization Set

What is a Characterization Set ?

- Let $\mathcal{M}$=(I, O, S, s_0, f, g) be a ***minimal*** and ***complete*** FSM
- W is a finite set of input sequences that distinguish the behavior of any pair of states in $\mathcal{M}$.
  - Each input sequence in W is of finite length
- Given states s_i and $s_j \in S$,
  W contains a string α such that $g(s_i, \alpha) \neq g(s_j, \alpha)$

An Example of W

- For the following $\mathcal{M}$, we have W={baaa,aa,aaa}
- For example, baaa distinguishes state s_1 from s_2 as $g(s_1, baaa) \neq g(s_2, baaa)$ More precisely, $g(s_1, baaa) = 1101$ and $g(s_2, baaa) = 1100$



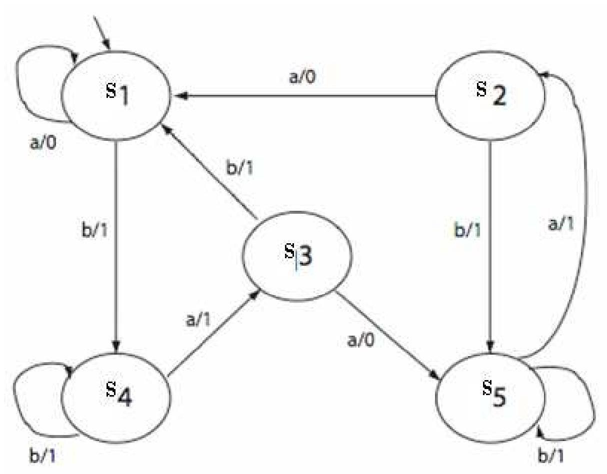
Step 2.1: Construction of the K-Equivalence Partitions

What is a k-equivalence partition of S?

- Given an FSM $\mathcal{M}$=(I, O, S, s_0, f, g) *k*-equivalence partition of S, denoted as P_k, is a collection of *n* finite sets Σ_{k1}, Σ_{k2} … Σ_{kn} such that
  - $\cup^{n}_{i=1} \Sigma_{ki} = S$
  - states in Σ_{ki} for $1 \leq i \leq n$ are *k*-equivalent
  - If state *u* is in Σ_{ki} and *v* in Σ_{kj} for $i \neq j$, then *u* and *v* are *k*-distinguishable

Construction of One-Equivalence Partition (1)

- Computing the one-equivalence partition, P_1, for the following FSM
- Start with a tabular representation of $\mathcal{M}$



Current state	Output		Next state	
	a	b	a	b
s1	0	1	s1	s4
s2	0	1	s1	s5
s3	0	1	s5	s1
s4	1	1	s3	s4
s5	1	1	s2	s5

Construction of One-Equivalence Partition (2)

- Group states identical in their *output* entries. This gives us 1-partition P_1 consisting of $\Sigma_1=\{s_1, s_2, s_3\}$ and $\Sigma_2 =\{s_4, s_5\}$

P_1 Table

Σ	Current state	Output		Next state	
		a	b	a	b
1	s1	0	1	s1	s4
	s2	0	1	s1	s5
	s3	0	1	s5	s1
2	s4	1	1	s3	s4
	s5	1	1	s2	s5

- We have the one-equivalence partition as follows
  - $P_1 = \{1, 2\}$
  - Group 1 = $\Sigma_{11} = \{s_1, s_2, s_3\}$
  - Group 2 = $\Sigma_{12} = \{s_4, s_5\}$

Construction of Two-Equivalence Partition (1)

- **Rewrite P_1 Table**. Remove the output columns. Replace a state entry s_i by s_{ij} where j is the group number in which lies state s_i

Σ	Current state	Next state	
		a	b
1	s1	s11	s42
	s2	s11	s52
	s3	s52	s11
2	s4	s31	s42
	s5	s21	s52

group number

state s_5 is in group 2

state s_1 is in group 1

Construction of Two-Equivalence Partition (2)

- **Construct P_2 Table**. Group all entries with *identical second subscripts* under the *next state column*. This gives us the P_2 table.
  - Note the change in second subscripts
  - We have three groups in the P_2 table

P_2 Table

Σ	Current state	Next state	
		a	b
1	s1	s11	s43
	s2	s11	s53
2	s3	s53	s11
3	s4	s32	s43
	s5	s21	s53

state s_5 is in group 3

Construction of Three-Equivalence Partition

- **Construct P_3 Table**. Group all entries with *identical second subscripts* under the next state column. This gives us the P_3 table.
  - Note the change in second subscripts
  - We have four groups in the P_3 table

P_3 Table

Σ	Current state	Next state	
		a	b
1	s1	s11	s43
	s2	s11	s54
2	s3	s54	s11
3	s4	s32	s43
4	s5	s21	s54

state s_5 is in group 4

Construction of Four-Equivalence Partition

- **Construct P_4 Table**. Continuing with regrouping and re-labeling
  - Note the change in second subscripts
  - We have five groups in the P_4 table

P_4 Table

Σ	Current state	Next state	
		a	b
1	s1	s11	s44
2	s2	s11	s55
3	s3	s55	s11
4	s4	s33	s44
5	s5	s22	s55

state s_5 is in group 5

k-equivalence partition: Convergence

- The process is guaranteed to converge
- When the process converges, and the machine is minimal, each state will be in a separate group
- The next step is to obtain the *distinguishing strings for each state*

Step 2.2: Using the W-Procedure to derive the Characterization Set W

A procedure to derive W from a set of partition tables constructed at Step 2.1

The W-Procedure (1)

- Let $\mathcal{M}$={S, I, O, f, g, s_o} be the FSM for which P = {P_1, P_2, …P_n} is the *set of k-equivalence partition tables* for k = 1, 2, …, *n*
  Initialize W = ∅
- Traverse the *k*-equivalence partitions in reverse order to obtain distinguishing sequence for each pair of states

Finding the distinguishing sequences (1)

- Let us find a distinguishing sequence for states s_1 and s_2
- Find Tables P_i and P_{i+1} such that (s_1, s_2) are in the same group in P_i and different groups in P_{i+1}. We get P_3 and P_4
- Initialize z=ε. Find the input symbol that distinguishes s_1 and s_2 in P_3. This symbol is b. We update z to z.b. Hence z now becomes b.

Finding the distinguishing sequences (2)

- The next states for s_1 and s_2 on *b* are, respectively, s_4 and s_5.
- We find that s_4 and s_5 are in the same group in P_2 and different groups in P_3.
- We move to the P_2 table and find the input symbol that distinguishes s_4 and s_5. Let us select a as the distinguishing symbol. Update z which now becomes ba.
- Refer to Table P_2, the next states for states s_4 and s_5 on symbol a are, respectively, s_3 and s_2. These two states are distinguished in P_1 by a and b (i.e., s_2 and s_3 are in the same group in P_1 and different groups in P_2). Let us select a. We update z to baa.

Finding the distinguishing sequences (3)

- Refer to Table P_1. The next states for s_2 and s_3 on a are, respectively, s_1 and s_5.
- We find that s_1 and s_5 are in the same group in the original table and different groups in P_1.
- Moving to the original state transition table we obtain a as the distinguishing symbol for s_1 and s_5
- We update z to baaa. This is the *farthest we can go backwards* through the various tables. baaa is the desired distinguishing sequence for states s_1 and s_2. Check that $g(s_1, baaa) = 1101$ and $g(s_2, baaa) = 1100$. We have $g(s_1, baaa) \neq g(s_2, baaa)$

Finding the distinguishing sequences (4)

- Using the procedure analogous to the one used for s_1 and s_2, we can find the distinguishing sequence for each pair of states. This leads us to the following characterization set for our FSM

- We have the distinguishing sequences as follows
  - $s_1, s_2 \rightarrow$ baaa
  - $s_1, s_3 \rightarrow$ aa
  - $s_1, s_4 \rightarrow$ a
  - $s_1, s_5 \rightarrow$ a
  - $s_2, s_3 \rightarrow$ aa

  $s_2, s_4 \rightarrow$ a
  $s_2, s_5 \rightarrow$ a
  $s_3, s_4 \rightarrow$ a
  $s_3, s_5 \rightarrow$ a
  $s_4, s_5 \rightarrow$ aaa

  This gives W={a, aa, aaa, baaa}

Where Are We?

- Step 1: Estimate the maximum number of states (*m*) in the correct implementation of the given FSM $\mathcal{M}$
- Step 2: Construct the characterization set W for $\mathcal{M}$
- Step 3: (a) Construct the *testing tree* for $\mathcal{M}$ and (b) generate the transition cover set P from the testing tree
- Step 4: Construct set Z from W and *m*
- Step 5: Desired test se is P.Z

Step 3.1: Construction of the Testing Tree for $\mathcal{M}$

Tree Construction (1)

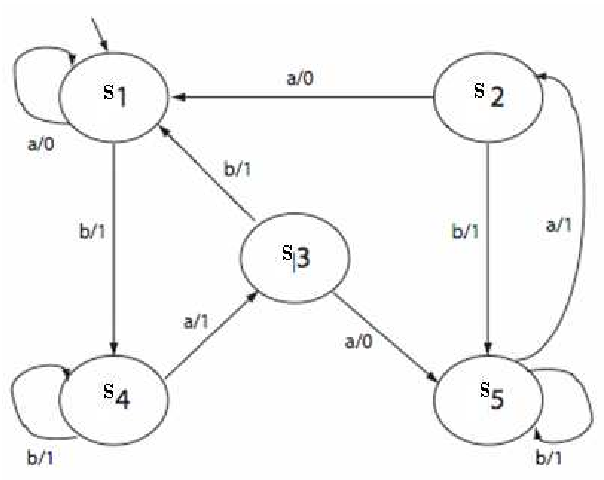
- A testing tree of an FSM, $\mathcal{M}$={S, I, O, f, g, s_o}, is a tree *rooted at the initial state*. It contains *at least one path from the initial state to the remaining states* in $\mathcal{M}$.

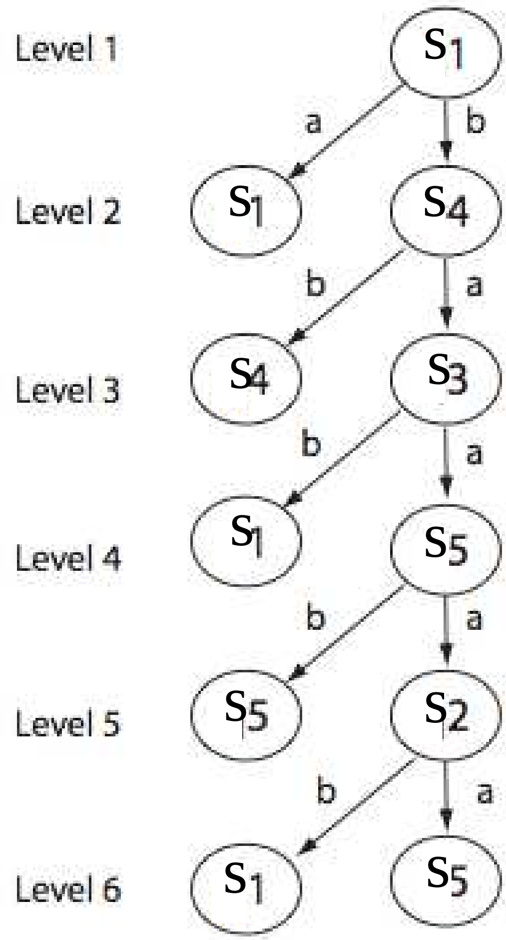
- Construction procedure

  - State s_0, the initial state, is the root of the testing tree.
    Suppose that the testing tree has been constructed until level *k*
    The (k+1)th level is built as follows

  - Select a node *n* at level *k*.
    If *n* appears at any level from 1 through *k* , then *n* is a leaf node and is not expanded any further.
    If *n* is not a leaf node then we expand it by adding a branch from node *n* to a new node *m* if $f(n, \alpha)=m$ for $\alpha \in$ I.
    This branch is labeled as α.
    This step is repeated for all nodes at level *k*.

Tree Construction (2)





Start here, initial state is the root

s_1 becomes leaf, s_4 can be expanded

No further expansion possible

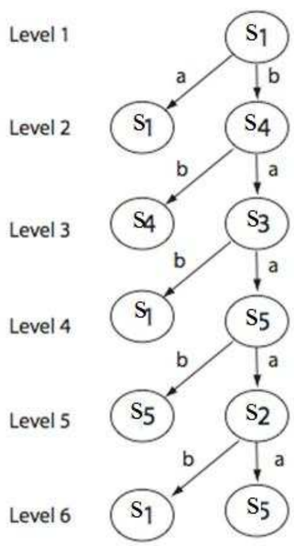
Tree Construction (3)

- The testing tree is initialized with the initial state s_1 as the root node.
- This is level 1 of the tree.
- We note that $g(s_1, a) = s_1$ and $g(s_1, b) = s_{4.}$ Hence, we create two nodes at the next level and label them as s_1 and s_4.
- The branches from s_1 to s_1 and s_4 are labeled, respectively, a and b.
- As s_1 is the only node at level 1, we now proceed to expand the tree to form level 2.
- At level 2, we first consider the node labeled s_1. However, another mode labeled s_1 already appears at level 1; hence, this node becomes a leaf node and is not expanded any further.
- Next, we examine the node labeled s_4. We note that $g(s_4, a) = s_3$ and $g(s_4, b) = s_{4.}$ We, therefore, create two new nodes at level 3 and label these as s_3 and s_4 and label the corresponding branches as a and b, respectively.

Step 3.2: Generation of the transition cover set P from the testing tree

Find The Transition Cover Set from a Testing Tree

- A transition cover set P is a set of all strings *representing sub-paths*, starting at the root, *in the testing tree*.
  - A sub-path is a path starting from the root of the testing tree and terminating in any node of the tree.
  - Concatenation of the labels along the edges of a sub-path is a string that belongs to P.
  - The empty string (ε) also belongs to P.



P={ε, a, b, bb, ba, bab, baa, baab, baaa, baaab, baaaa}

Why is it called a Transition Cover Set?

- Exciting an FSM in s_0, the initial set, with an element of P, forces the FSM into some state.
- After the FSM has been excited with all elements of P, each time starting in the initial state, the FSM has reached every state.
- Thus, exciting an FSM with elements of P ensures that ***all states are reached*** and ***all transitions have been traversed at least once***.
  - The empty input sequence does not traverse any branch but is useful in constructing some desired test sequence.

Step 4: Construction of Z from W and *m*

Construction of Z from W and *m*

- Given that I is the input alphabet and W the characterization set. Suppose that the number of states estimated to be in the implementation under test is m, and the number of states in the design specification is n, $m > n$.

- We compute Z as

$$Z = (I^0.W) \cup (I^1.W) \cup \ldots.. (I^{m-1-n}.W) \cup (I^{m-n}.W)$$

  - Recall that $I^0 = \{\varepsilon\}$, $I^1 = I$, $I^2 = I . I$, and so on, where (.) denotes string concatenation.

- For $m = n$, we get $Z = I^0.W=W$

- For $m < n$, we use Z = IW

Step 5.1: Test Generation

Generation of a Test Set from P and Z

- The test inputs based on the given FSM $\mathcal{M}$ can now be derived as T=P.Z
- Let's use the same example. Suppose $m = n = 5$. We also have I={a, b} and W (the characterization set) ={a, aa, aaa, baaa} (see slide 46)
  - Concatenating P with Z, we obtain the desired test set as follows
    - Z = I^0.W ={a, aa, aaa, baaa} (Note: I^0 = {ε} see slide 56)
    - T = P . Z ={ε, a, b, bb, ba, bab, baa, baab, baaa, baaab, baaaa} . {a, aa, aaa, baaa}
- If we assume that the implementation has one extra state, that is, $m = 6$, then we have
    - Z = I^0.W ∪ (I^1.W) ={a, aa, aaa, baaa, aa, aaa, aaaa, abaaa, ba, baa, baaa, bbaaa}
    - T = P . Z ={ε, a, b, bb, ba, bab, baa, baab, baaa, baaab, baaaa} .
      {a, aa, aaa, baaa, aa, aaa, aaaa, abaaa, ba, baa, baaa, bbaaa}

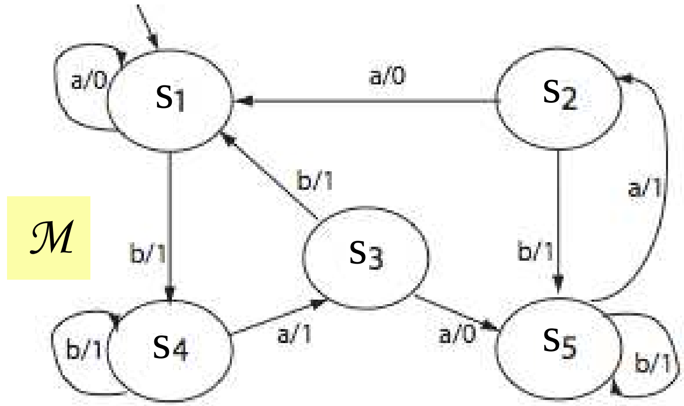
Step 5.2: Test Execution

Execution of the Generated Test Set

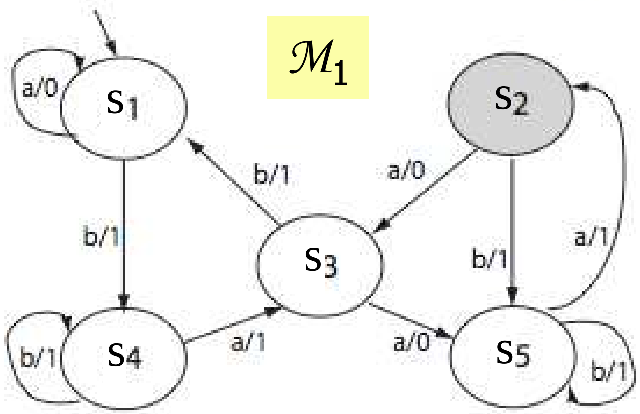
- To test the given implementation against its specification M, we do the following for each test input
  - Find the expected response to each element of T
  - Generate test cases for the application. Note that even though the application is modeled by $\mathcal{M}$, there might be variables to be set before it can be exercised with elements of T.
  - Execute the application and check if the response matches.
    Reset the application to the initial state after each test.
    - **A mismatch between the expected and the actual response does not necessarily imply an error in the implementation.**
      - Is the specification error free?
      - Are the expected and actual responses determined without any error?
      - Is the comparison between them correct?

      If the answer is YES to all these questions, then a mismatch implies an error in the implementation.

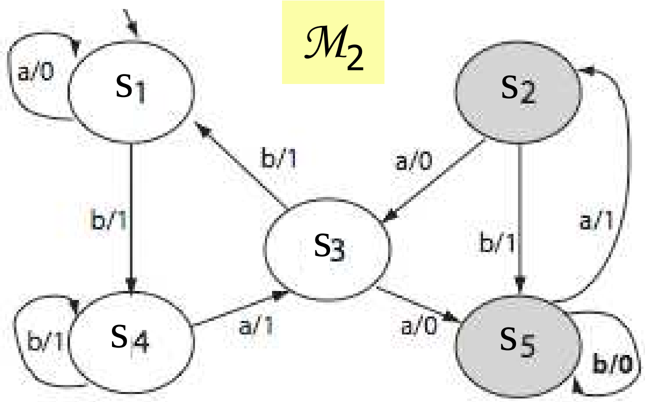
Example One (1): *n* = *m* = 5



(a) Correct design



(b) Transfer error in state s_2

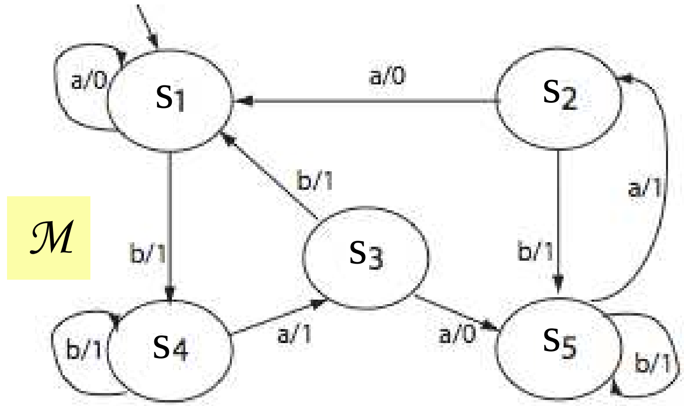


(b) Transfer error in state s_2 and operation error in state s_5

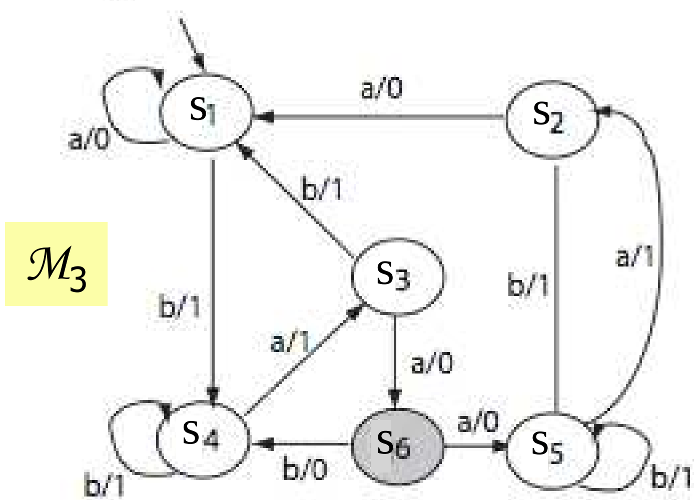
Example One (2): $n = m = 5$

- To test $\mathcal{M}_1$ against $\mathcal{M}$, we apply each test t from the set P.Z and compare $\mathcal{M}(t)$ with $\mathcal{M}_1(t)$.
  - We find that when ***t* = baaaaaa**, $\mathcal{M}(t) = 1101000$ and $\mathcal{M}_1(t) = 1101001$. Hence, the input sequence baaaaaa reveals the transfer error in $\mathcal{M}_1$.
- Similarly, to test $\mathcal{M}_2$ against $\mathcal{M}$, we apply each test t from the set P.Z and compare $\mathcal{M}(t)$ with $\mathcal{M}_2(t)$.
  - We find that when ***t* = baaba**, $\mathcal{M}(t) = 11011$ and $\mathcal{M}_2(t) = 11001$. Hence, the input sequence baaba reveals the transfer error in $\mathcal{M}_2$.

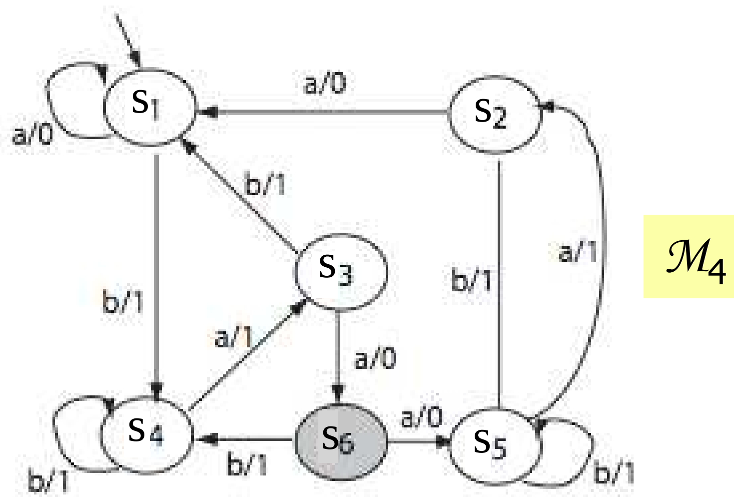
Example Two (1): *n* = 5 & *m* = 6



(a) Correct design



(d) an extra state



(e) an extra state

Example Two (2): *n* = 5 & *m* = 6

- To test $\mathcal{M}_3$ against $\mathcal{M}$, we apply each test *t* from the set P.Z and compare $\mathcal{M}(t)$ with $\mathcal{M}_3(t)$.
  - We find that when ***t* = baaba**, $\mathcal{M}(t)$ = 11011 and $\mathcal{M}_3(t)$ = 11001.
    Hence, the input sequence baaba reveals the transfer error in $\mathcal{M}_3$.
- Similarly, to test $\mathcal{M}_4$ against $\mathcal{M}$, we apply each test *t* from the set P.Z and compare $\mathcal{M}(t)$ with $\mathcal{M}_4(t)$.
  - We find that when ***t* = *baaa***, $\mathcal{M}(t)$ = 1101 and $\mathcal{M}_4(t)$ = 1100.
    Hence, the input sequence baaa reveals the transfer error in $\mathcal{M}_4$.

Test Sets Generated Using the W-Method

- The W-method is used for *constructing a test set* from a given FSM $\mathcal{M}$.
- The test set so constructed is *a finite set of sequences* that can be input to a *program* whose *control structure is modeled by* $\mathcal{M}$.
- The tests can also be input to a *design* to test its correctness with respect to some specification.
- Most software systems *cannot be modeled 100% accurately* using an FSM. However, the *global control structure* of a software system cam be modeled by an FSM.
- Tests generated using the W-method, or any other method based exclusively on a finite-state model of an implementation is likely to *reveal only certain types of faults.* (see slide 4)

Automata Theoretical

versus

Control-Flow-Based Techniques

Question

- The W method is an *automata theoretic* method for test generation.
- In contrast, many books on software testing mention *control flow-based techniques* for test generation.
- What is the difference between the two types of techniques and their fault detection abilities?

Control-Flow-Based Techniques (1)

- State cover: A test set T is considered adequate with respect to the *state cover* criterion for an FSM $\mathcal{M}$ if the execution of $\mathcal{M}$ against each element of T causes each state in $\mathcal{M}$ to be visited at least once
- Transition cover: A test set T is considered adequate with respect to the *branch/transition cover* criterion for an FSM $\mathcal{M}$ if the execution of $\mathcal{M}$ against each element of T causes each transition in $\mathcal{M}$ to be taken at least once
- Switch cover: A test set T is considered adequate with respect to the *1-switch cover* criterion for an FSM $\mathcal{M}$ if the execution of $\mathcal{M}$ against each element of T causes each pair of transitions (tr_1, tr_2) in $\mathcal{M}$ to be taken at least once, where for some input substring *ab* tr_1: $f(s_j, a)=s_i$ and tr_2: $f(s_i, b) = s_k$

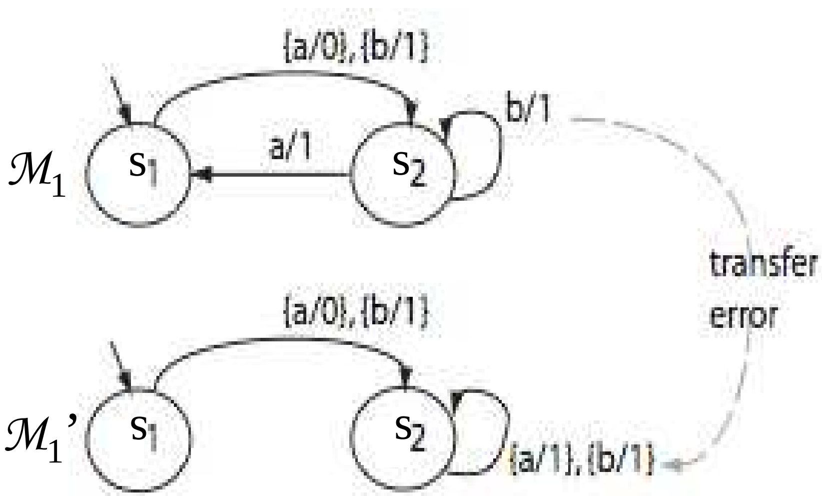
Control-Flow-Based Techniques (2)

- Boundary interior cover: A test set T is considered adequate with respect to the *boundary-interior cover* criterion for an FSM $\mathcal{M}$ if the execution of $\mathcal{M}$ against each element of T causes *each loop (a self-transition) across states* to be traversed *zero times and at least once*.
  - Exiting the loop upon arrival covers the "*boundary*" condition
  - Entering it and traversing the loop at least once covers the "*interior*" condition

- The following examples illustrate weaknesses of the state cover, branch/transition cover, switch cover and the boundary-interior cover test-adequacy criteria.

Control-Flow-Based Techniques (3)

- Consider the following machines, a correct one ($\mathcal{M}_1$) and one with a transfer error ($\mathcal{M}_1$')
- t=abba *covers all states* but does not reveal the error. Both machines generate the same output which is 0111.
- Will the tests generated by the W method reveal this error?
  - Check it out!



Refer to Example 3.38, Page 263

Control-Flow-Based Techniques (4)

- Consider the following machines, a correct one ($\mathcal{M}_2$) and one with a transfer error ($\mathcal{M}_2$')
- There are 12 branch pairs, such as (tr_1, tr_2), (tr_1, tr_3), tr_6, tr_5)
- Consider the test set: {bb, baab, aabb, aaba, abbaab}
  - Does it cover all branches?

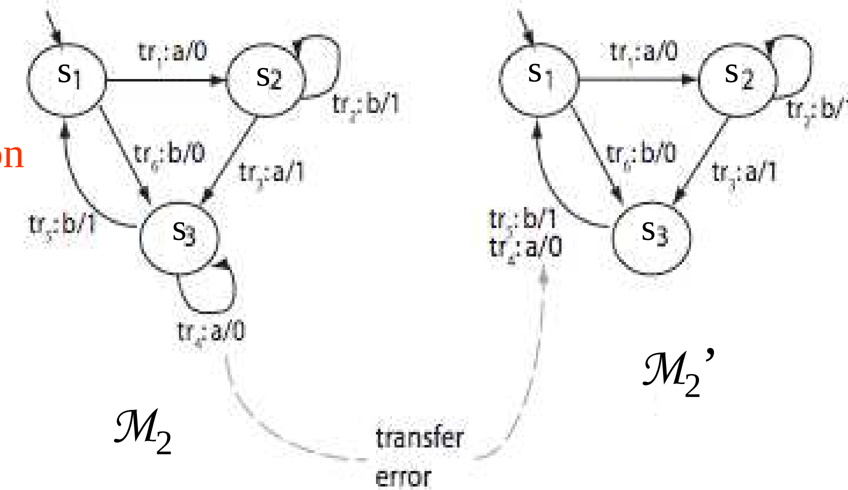
    Yes

    T also satisfies the 1-switch cover criterion
  - Does it reveal the error?

    No
  - Are the states in M_2 1-distinguishable?

    YES

    For each pair of states (s_i, s_j), $i \neq j$, there exists a string of length 1 that distinguishes s_i from s_j

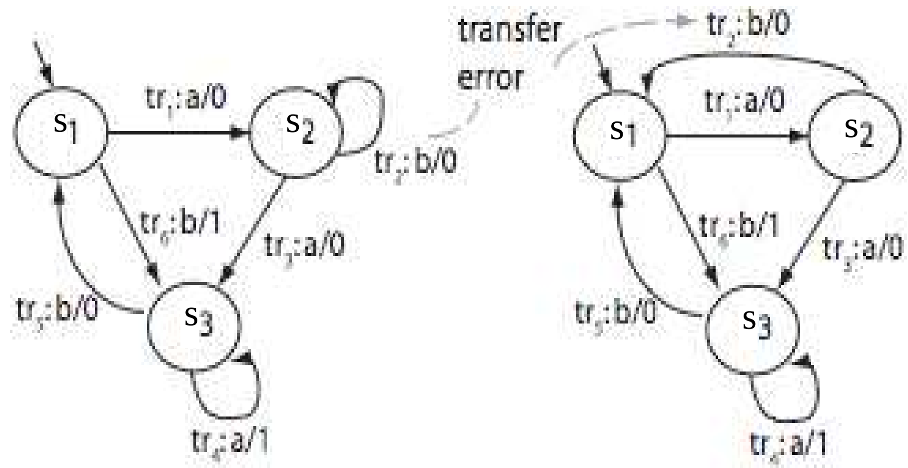


Refer to Example 3.38

Control-Flow-Based Techniques (5)

- Consider the following machines, a correct one ($\mathcal{M}_3$) and one with a transfer error ($\mathcal{M}_3$’)
  - There are two loops in $\mathcal{M}_3$ one in state s_2 and the other in s_3

- Consider T={t_1: aab, t_2: abaab}
  - T is adequate with respect to the boundary-interior cover criterion
  - t_1 causes both loops to exit without looping in either s_2 or s_3
  - t_2 causes each loop to be traversed once



$\mathcal{M}_3$ $\mathcal{M}_3$’

- Is the error revealed by T?
  - $g_{\mathcal{M}3}(s_1, t_1) = g_{\mathcal{M}'3}(s_1, t_1) = 000$
  - $g_{\mathcal{M}3}(s_1, t_2) = g_{\mathcal{M}'3}(s_1, t_2) = 00010$
  - T cannot distinguish $\mathcal{M}_3$ from $\mathcal{M}'_3$ and hence does not reveal the error

Summary

- Behavior of a large variety of applications can be modeled using finite state machines (FSM)
- The W method is an automata theoretic method to generate tests from a given FSM model
- Tests generated by using the W method are guaranteed to detect all operation errors, transfer errors, and missing/extra state errors in the implementation given that the FSM representing the implementation is complete, connected, and minimal.
  - What happens if it is not?
- Automata theoretic techniques generate tests superior in their fault detection ability than their control-theoretic counterparts