UNIT V
STATES, STATE GRAPHS, AND TRANSITION TESTING

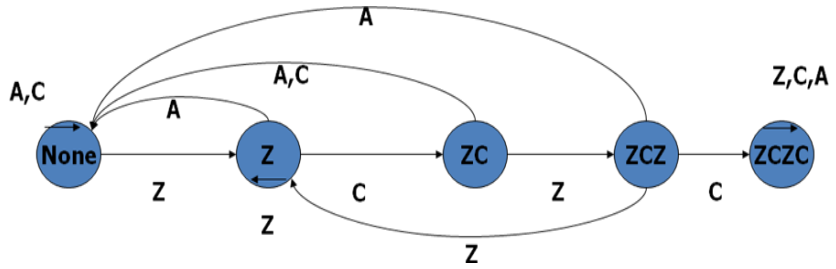
Introduction

- The finite state machine is as fundamental to software engineering as boolean algebra to logic.
- State testing strategies are based on the use of finite state machine models for software structure, software behavior, or specifications of software behavior.
- Finite state machines can also be implemented as table-driven software, in which case they are a powerful design option.

State Graphs

- A state is defined as: "A combination of circumstances or attributes belonging for the time being to a person or thing."
- For example, a moving automobile whose engine is running can have the following states with respect to its transmission.
  - Reverse gear
  - Neutral gear
  - First gear
  - Second gear
  - Third gear
  - Fourth gear

  State graph - Example
- For example, a program that detects the character sequence "ZCZC" can be in the following states.
- Neither ZCZC nor any part of it has been detected.
  - Z has been detected.
  - ZC has been detected.
  - ZCZ has been detected.
  - ZCZC has been detected.



States are represented by Nodes. State are numbered or may identified by words or whatever else is convenient.

Inputs and Transitions

- Whatever is being modeled is subjected to inputs. As a result of those inputs, the state changes, or is said to have made a Transition.
- Transitions are denoted by links that join the states.
- The input that causes the transition are marked on the link; that is, the inputs are link weights.
- There is one out link from every state for every input.

- If several inputs in a state cause a transition to the same subsequent state, instead of drawing a bunch of parallel links we can abbreviate the notation by listing the several inputs as in: “input1, input2, input3……”.

Finite State Machine

- A finite state machine is an abstract device that can be represented by a state graph having a finite number of states and a finite number of transitions between states.
    - Outputs
- An output can be associated with any link.
- Out puts are denoted by letters or words and are separated from inputs by a slash as follows: “input/output”.
- As always, output denotes anything of interest that’s observable and is not restricted to explicit outputs by devices.
- Outputs are also link weights.
- If every input associated with a transition causes the same output, then denoted it as:
    - “input1, input2, input3……../output”

State Tables

- Big state graphs are cluttered and hard to follow.
- It’s more convenient to represent the state graph as a table (the state table or state transition table) that specifies the states, the inputs, the transitions and the outputs.
- The following conventions are used:
- Each row of the table corresponds to a state.
- Each column corresponds to an input condition.
- The box at the intersection of a row and a column specifies the next state (the transition) and the output, if any.

State Table-Example

inputs

STATE	Z	C	A
NONE	Z	NONE	NONE
Z	Z	ZC	NONE
ZC	ZCZ	NONE	NONE
ZCZ	Z	ZCZC	NONE
ZCZC	ZCZC	ZCZC	ZCZC

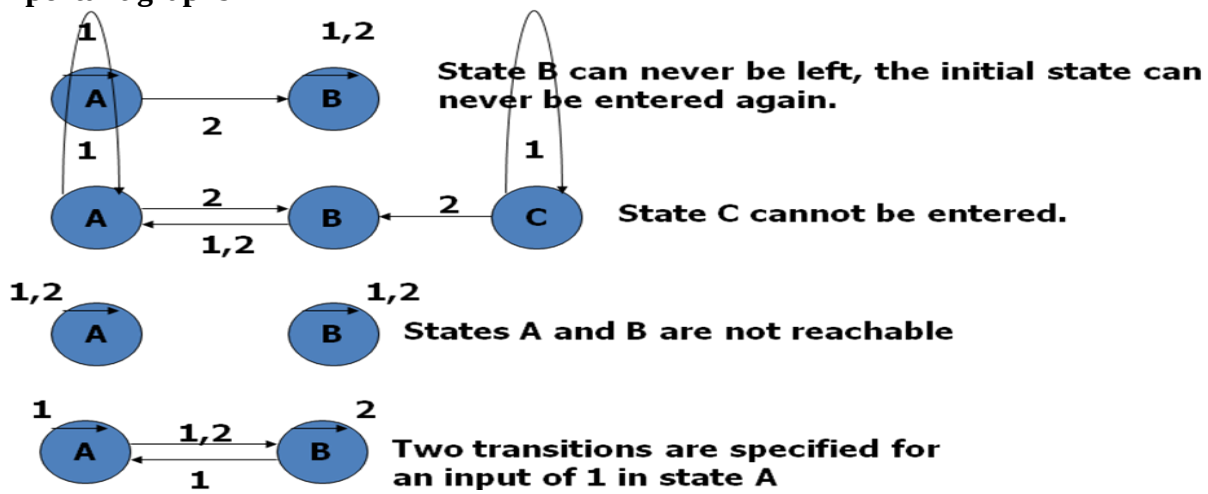
Time Versus Sequence

- State graphs don't represent time-they represent sequence.
- A transition might take microseconds or centuries;
- A system could be in one state for milliseconds and another for years- the state graph would be the same because it has no notion of time.
- Although the finite state machines model can be elaborated to include notions of time in addition to sequence, such as time Petri Nets.
    - Software implementation
- There is rarely a direct correspondence between programs and the behavior of a process described as a state graph.
- The state graph represents, the total behavior consisting of the transport, the software, the executive, the status returns, interrupts, and so on.
- There is no simple correspondence between lines of code and states. The state table forms the basis.

Good State Graphs and Bad

- What constitutes a good or a bad state graph is to some extent biased by the kinds of state graphs that are likely to be used in a software test design context.
- Here are some principles for judging.
    - The total number of states is equal to the product of the possibilities of factors that make up the state.
    - For every state and input there is exactly one transition specified to exactly one, possibly the same, state.
    - For every transition there is one output action specified. The output could be trivial, but at least one output does something sensible.
    - For every state there is a sequence of inputs that will drive the system back to the same state.

Important graphs

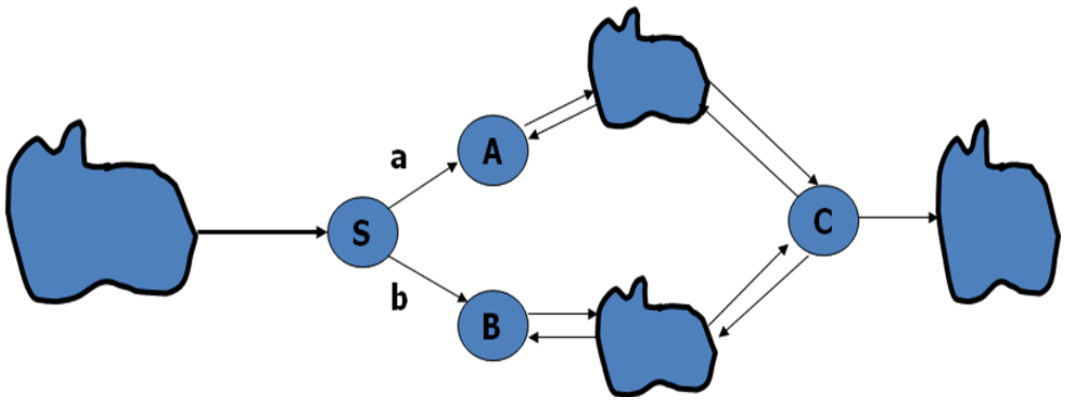


State Bugs-Number of States

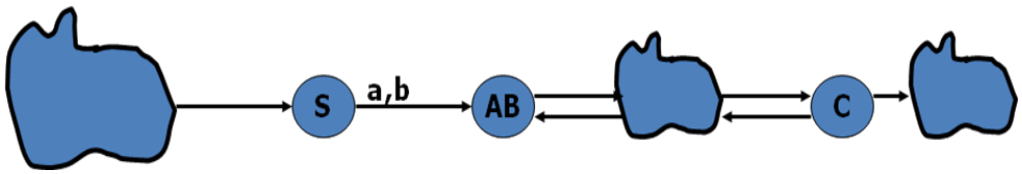
- The number of states in a state graph is the number of states we choose to recognize or model.
- The state is directly or indirectly recorded as a combination of values of variables that appear in the data base.
- For example, the state could be composed of the value of a counter whose possible values ranged from 0 to 9, combined with the setting of two bit flags, leading to a total of 2*2*10=40 states.
- The number of states can be computed as follows:
  - Identify all the component factors of the state.
  - Identify all the allowable values for each factor.
  - The number of states is the product of the number of allowable values of all the factors.
- Before you do anything else, before you consider one test case, discuss the number of states you think there are with the number of states the programmer thinks there are.
- There is no point in designing tests intended to check the system's behavior in various states if there's no agreement on how many states there are.
  - Impossible States
- Some times some combinations of factors may appear to be impossible.
- The discrepancy between the programmer's state count and the tester's state count is often due to a difference of opinion concerning "impossible states".
- A robust piece of software will not ignore impossible states but will recognize them and invoke an illogical condition handler when they appear to have occurred.

Equivalent States

- Two states are Equivalent if every sequence of inputs starting from one state produces exactly the same sequence of outputs when started from the other state. This notion can also be extended to set of states.



Merging of Equivalent States



Recognizing Equivalent States

- Equivalent states can be recognized by the following procedures:
- The rows corresponding to the two states are identical with respect to input/output/next state but the name of the next state could differ.
- There are two sets of rows which, except for the state names, have identical state graphs with respect to transitions and outputs. The two sets can be merged.

TransitionBugs-

unspecified and contradictory Transitions

- Every input-state combination must have a specified transition.
- If the transition is impossible, then there must be a mechanism that prevents the input from occurring in that state.
- Exactly one transition must be specified for every combination of input and state.
- A program can't have contradictions or ambiguities.
- Ambiguities are impossible because the program will do something for every input. Even the state does not change, by definition this is a transition to the same state.

Unreachable States

- An unreachable state is like unreachable code.
- A state that no input sequence can reach.
- An unreachable state is not impossible, just as unreachable code is not impossible
- There may be transitions from unreachable state to other states; there usually because the state became unreachable as a result of incorrect transition.
- There are two possibilities for unreachable states:
    - There is a bug; that is some transitions are missing.
    - The transitions are there, but you don't know about it.

Dead States

- A dead state is a state that once entered cannot be left.
- This is not necessarily a bug but it is suspicious.

Output Errors

- The states, transitions, and the inputs could be correct, there could be no dead or unreachable states, but the output for the transition could be incorrect.
- Output actions must be verified independently of states and transitions.

State Testing

Impact of Bugs

- If a routine is specified as a state graph that has been verified as correct in all details. Program code or table or a combination of both must still be implemented.
- A bug can manifest itself as one of the following symptoms:
- Wrong number of states.
- Wrong transitions for a given state-input combination.
- Wrong output for a given transition.
- Pairs of states or sets of states that are inadvertently made equivalent.
- States or set of states that are split to create in equivalent duplicates.

- States or sets of states that have become dead.
- States or sets of states that have become unreachable.

Principles of State Testing

- The strategy for state testing is analogous to that used for path testing flow graphs.
- Just as it's impractical to go through every possible path in a flow graph, it's impractical to go through every path in a state graph.
- The notion of coverage is identical to that used for flow graphs.
- Even though more state testing is done as a single case in a grand tour, it's impractical to do it that way for several reasons.
- In the early phases of testing, you will never complete the grand tour because of bugs.
- Later, in maintenance, testing objectives are understood, and only a few of the states and transitions have to be tested. A grand tour is waste of time.
- Theirs is no much history in a long test sequence and so much has happened that verification is difficult.

Starting point of state testing

- Define a set of covering input sequences that get back to the initial state when starting from the initial state.
- For each step in each input sequence, define the expected next state, the expected transition, and the expected output code.
- A set of tests, then, consists of three sets of sequences:
    - Input sequences
    - Corresponding transitions or next-state names
    - Output sequences

Limitations and Extensions

- State transition coverage in a state graph model does not guarantee complete testing.
- How defines a hierarchy of paths and methods for combining paths to produce covers of state graphs.
- The simplest is called a "0 switch" which corresponds to testing each transition individually.
- The next level consists of testing transitions sequences consisting of two transitions called "1 switches".
- The maximum length switch is "n-1 switch" where there are n numbers of states.
    - Situations at which state testing is useful
- Any processing where the output is based on the occurrence of one or more sequences of events, such as detection of specified input sequences, sequential format validation, parsing, and other situations in which the order of inputs is important.
- Most protocols between systems, between humans and machines, between components of a system.
- Device drivers such as for tapes and discs that have complicated retry and recovery procedures if the action depends on the state.

Whenever a feature is directly and explicitly implemented as one or more state transition tables.

GRAPH MATRICES AND APPLICATIONS

Problem with Pictorial Graphs

- Graphs were introduced as an abstraction of software structure.
- Whenever a graph is used as a model, sooner or later we trace paths through it- to find a set of covering paths, a set of values that will sensitize paths, the logic function that controls the flow, the processing time of the routine, the equations that define the domain, or whether a state is reachable or not.
- Path is not easy, and it's subject to error. You can miss a link here and there or cover some links twice.
- One solution to this problem is to represent the graph as a matrix and to use matrix operations equivalent to path tracing. These methods are more methodical and mechanical and don't depend on your ability to see a path they are more reliable.

Tool Building

- If you build test tools or want to know how they work, sooner or later you will be implementing or investigating analysis routines based on these methods.
- It is hard to build algorithms over visual graphs so the properties or graph matrices are fundamental to tool building.

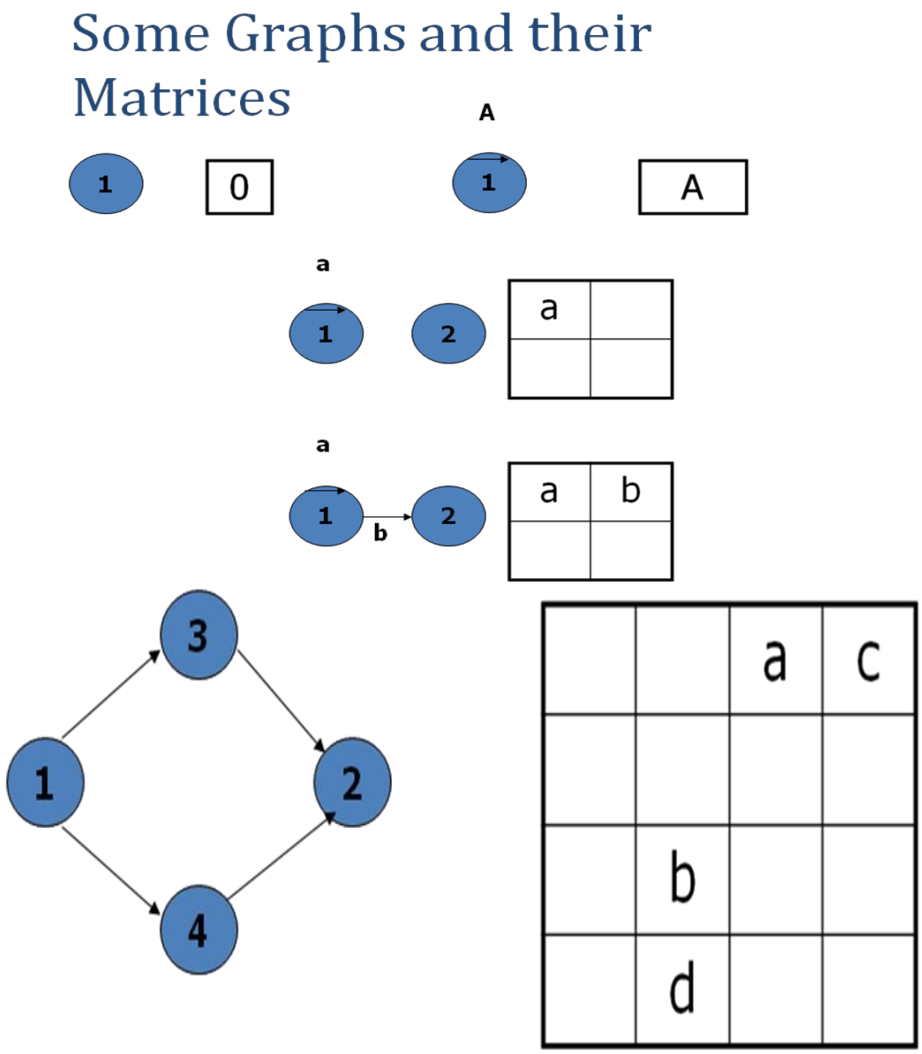
The Basic Algorithms

- The basic tool kit consists of:
- Matrix multiplication, which is used to get the path expression from every node to every other node.
- A partitioning algorithm for converting graphs with loops into loop free graphs or equivalence classes.
- A collapsing process which gets the path expression from any node to any other node.

The Matrix of a Graph

- A graph matrix is a square array with one row and one column for every node in the graph.
- Each row-column combination corresponds to a relation between the node corresponding to the row and the node corresponding to the column.
- The relation for example, could be as simple as the link name, if there is a link between the nodes.

- Some of the things to be observed:
- The size of the matrix equals the number of nodes.
- There is a place to put every possible direct connection or link between any and any other node.
- The entry at a row and column intersection is the link weight of the link that connects the two nodes in that direction.
- A connection from node i to j does not imply a connection from node j to node i.
- If there are several links between two nodes, then the entry is a sum; the "+" sign denotes parallel links as usual.



A simple weight

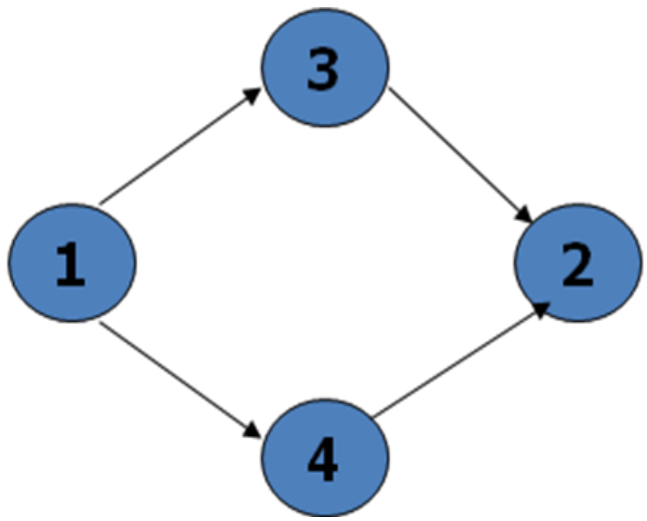
- A simplest weight we can use is to note that there is or isn't a connection. Let "1" mean that there is a connection and "0" mean that there isn't.
- The arithmetic rules are:
- 1+1=1 1*1=1
- 1+0=1 1*0=0
- 0+0=0 0*0=0
- A matrix defined like this is called connection matrix.

Connection matrix

- The connection matrix is obtained by replacing each entry with 1 if there is a link and 0 if there isn't.
- As usual we don't write down 0 entries to reduce the clutter.

		a	c
	b		
	d		

		1	1
	1		
	1		



Connection Matrix-continued

- Each row of a matrix denotes the out links of the node corresponding to that row.
- Each column denotes the in links corresponding to that node.
- A branch is a node with more than one nonzero entry in its row.
- A junction is node with more than one nonzero entry in its column.
- A self loop is an entry along the diagonal.

Cyclomatic Complexity

- The cyclomatic complexity obtained by subtracting 1 from the total number of entries in each row and ignoring rows with no entries, we obtain the equivalent number of decisions for each row. Adding these values and then adding 1 to the sum yields the graph's cyclomatic complexity.

		1	1	2-1=1
	1			1-1=0
	1			1-1=0

1+1=2 (cyclomatic complexity)

Relations

- A relation is a property that exists between two objects of interest.
- For example,
- “Node a is connected to node b” or aRb where “R” means “is connected to”.
- “a>=b” or aRb where “R” means greater than or equal”.
- A graph consists of set of abstract objects called nodes and a relation R between the nodes.
- If aRb, which is to say that a has the relation R to b, it is denoted by a link from a to b.
- For some relations we can associate properties called as link weights.

Transitive Relations

- A relation is transitive if aRb and bRc implies aRc.
- Most relations used in testing are transitive.
- Examples of transitive relations include: is connected to, is greater than or equal to, is less than or equal to, is a relative of, is faster than, is slower than, takes more time than, is a subset of, includes, shadows, is the boss of.
- Examples of intransitive relations include: is acquainted with, is a friend of, is a neighbor of, is lied to, has a du chain between.

Reflexive Relations

- A relation R is reflexive if, for every a, aRa.
- A reflexive relation is equivalent to a self loop at every node.
- Examples of reflexive relations include: equals, is acquainted with, is a relative of.
- Examples of irreflexive relations include: not equals, is a friend of, is on top of, is under.

Symmetric Relations

- A relation R is symmetric if for every a and b, aRb implies bRa.
- A symmetric relation mean that if there is a link from a to b then there is also a link from b to a.
- A graph whose relations are not symmetric are called directed graph.

- A graph over a symmetric relation is called an undirected graph.
- The matrix of an undirected graph is symmetric ($a_{ij}=a_{ji}$) for all i,j)

Antisymmetric Relations

- A relation R is antisymmetric if for every a and b, if aRb and bRa, then a=b, or they are the same elements.
- Examples of antisymmetric relations: is greater than or equal to, is a subset of, time.
- Examples of nonantisymmetric relations: is connected to, can be reached from, is greater than, is a relative of, is a friend of

quivalence Relations

- An equivalence relation is a relation that satisfies the reflexive, transitive, and symmetric properties.
- Equality is the most familiar example of an equivalence relation.
- If a set of objects satisfy an equivalence relation, we say that they form an equivalence class over that relation.
- The importance of equivalence classes and relations is that any member of the equivalence class is, with respect to the relation, equivalent to any other member of that class.
- The idea behind partition testing strategies such as domain testing and path testing, is that we can partition the input space into equivalence classes.
- Testing any member of the equivalence class is as effective as testing them all.

Partial Ordering Relations

- A partial ordering relation satisfies the reflexive, transitive, and antisymmetric properties.
- Partial ordered graphs have several important properties: they are loop free, there is at least one maximum element, and there is at least one minimum element.

The Powers of a Matrix

- Each entry in the graph's matrix expresses a relation between the pair of nodes that corresponds to that entry.
- Squaring the matrix yields a new matrix that expresses the relation between each pair of nodes via one intermediate node under the assumption that the relation is transitive.
- The square of the matrix represents all path segments two links long.
- The third power represents all path segments three links long.

Matrix Powers and Products

- Given a matrix whose entries are aij, the square of that matrix is obtained by replacing every entry with
    - n
    - $a_{ij}=\Sigma\ a_{ik}\ a_{kj}$
    - k=1
- more generally, given two matrices A and B with entries aik and bkj, respectively, their product is a new matrix C, whose entries are cij, where:
    - n
        - $C_{ij}=\Sigma\ a_{ik}\ b_{kj}$
        - k=1

Partitioning Algorithm

- Consider any graph over a transitive relation. The graph may have loops.
- We would like to partition the graph by grouping nodes in such a way that every loop is contained within one group or another.
- Such a graph is partially ordered.
- There are many used for an algorithm that does that:
- We might want to embed the loops within a subroutine so as to have a resulting graph which is loop free at the top level.
- Many graphs with loops are easy to analyze if you know where to break the loops.
- While you and I can recognize loops, it's much harder to program a tool to do it unless you have a solid algorithm on which to base the tool.

Node Reduction Algorithm (General)

- The matrix powers usually tell us more than we want to know about most graphs.
- In the context of testing, we usually interested in establishing a relation between two nodes- typically the entry and exit nodes.
- In a debugging context it is unlikely that we would want to know the path expression between every node and every other node.
- The advantage of matrix reduction method is that it is more methodical than the graphical method called as node by node removal algorithm.

1. Select a node for removal; replace the node by equivalent links that bypass that node and add those links to the links they parallel.
2. Combine the parallel terms and simplify as you can.
3. Observe loop terms and adjust the out links of every node that had a self loop to account for the effect of the loop.
4. The result is a matrix whose size has been reduced by 1. Continue until only the two nodes of interest exist.