State Graphs

→ The state graph & its associated state table are useful for describing software behaviour

→ Finite state machine is a functional testing tool & design programming tool

→ The finite state machine is a fundamental to software engineering as boolean algebra.

→ This machine can also be implemented as table driven software.

→ state testing strategies are based on the use of finite state machine model for software structure, software behaviour or specification of software behaviour.

→ states are denoted by nodes

Ex:1) A moving automobile engine is running can have the following states with respect to its transmission.
i) Reverse gear (ii) Neutral gear (iii) First gear
(iv) second gear (v) third gear (vi) fourth gear

Ex:2) A person checkbook can have these states with respect to bank balance
(i) equal (ii) less than (iii) Greater than

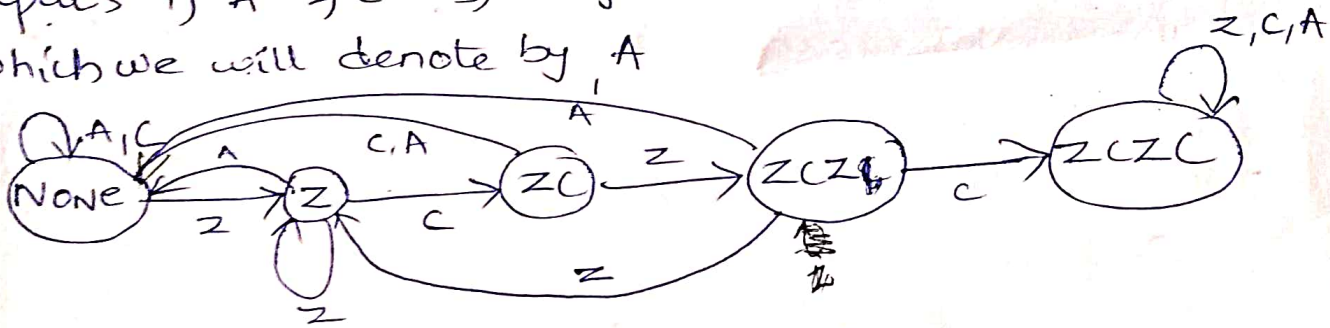
Ex:3) A word processing program menu can be in these states with respect to file manipulation
(i) create document (ii) Save document (iii) Rename - document (iv) copy document (v) Delete document

→ Transition: The Result of inputs, the state changes is said to be a transition.

→ The input that causes the transitions [illegible] the link i.e., the inputs are link weights, There is an outlink for every input.

→ Finite state machine is an abstract device that can be represented by a state graph having a finite no. of states & finite no. of transitions between states.

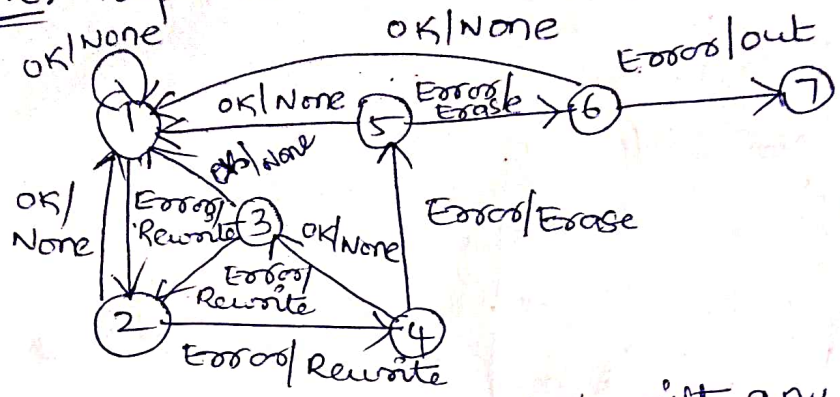
→ The ZCZC detection example can have the following inputs 1) A 2) C 3) Any character other than Z (or) C which we will denote by 'A'



The above state graph interpreted as

1) If the system is in the NONE state, any input other a 'z' will keep it in that state
2) If a 'z' received, the system transitions to the 'z' state
3) If the system is in the 'z' state & a 'z' is received, it will remain in the 'z' state. If a 'c' is received, it will go to the 'zc' state, if any other character is received it will go back to the NONE state because sequence has been broken
4) A 'z' received in the 'zc' state progresses to "zcz" state, but any other character breaks the sequence & return to "None" state
5) A 'c' received in the "zcz" state completes the sequence & the system enter the "zczc" state. if 'z' received it causes transition back to 'z' state, any other character causes a return to 'None' state.
6) The system stays in the "zczc" state, no matter what is received.

Ex: Tape control recovery routine state graph



→ output can be associated with any link outputs are separated from inputs input/output

→ If no errors are detected then input = OK and no special action is taken then output = None

→ If write error is detected then input = error backspace the tape one block & rewrite the block output = Rewrite

→ If rewrite is successful then input = OK

→ If there have been two successive rewrites & a third error occurs backspace & erase forward then output = erase

→ The inputs & actions have been simplified. There are only 2 inputs (OK, Error), & 4 outputs (Rewrite, Erase, None, out-of-service).

State table → we have to represent stategraph as a table.

→ state table (or) state transition table specifies, states inputs, transitions & outputs.

→ Each row of the table represents state

→ Each column of the table represents a condition

→ The intersection of a row & column specifies the next state & output.

Input

State	OK	Error
1	1/None	2/Rewrite
2	1/None	4/Rewrite
3	1/None	2/Rewrite
4	3/None	5/Erase
5	1/None	6/Erase
6	1/None	7/out
7	...	

→ state graph's don't represent time, represents sequence

Software Implementation

```
Begin
  present_state = Device_table (Device-name)
  Accept input_value
  Input_code = Input_code table (Input_value)
  pointer = Input_code (present state)
  New-state = transition_table (pointer)
  output-code = output_table (pointer)
  call output_handler (output_code)
  Device_table (Device-name) = New-state
End
```

→ state graph represents the total behaviour consists of transport, s/w, executive, status returns, interrupts.

→ There are 4 tables involved

1) A table (or) process that encodes the input values into

into a compact list (input-code-table)

2) A table that specifies next state for every combination of state & input code (transition table)

3) A table that specifies the output. (output-table)

4) " " " stores the present state of every device (or) process (Device-table).

→ The present state is fetched from memory. The input value is fetched. If it is already numerical it can be used directly otherwise it is encoded into a numerical value

→ The present state & input are combined to yield pointer of the transition table.

→ The output table either directly or via case statement contains a pointer to routine to executed for that state-input combination. The routine is invoked. The same pointer is used to fetch new state value which is then stored.

Good & Bad state graphs

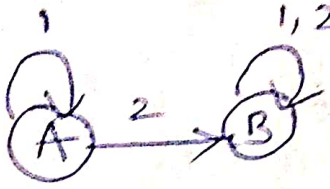
Principles of judging a graph as a good & bad state graph are

1) The total no. of states is equal to the product of the possibilities of factors that make up the state

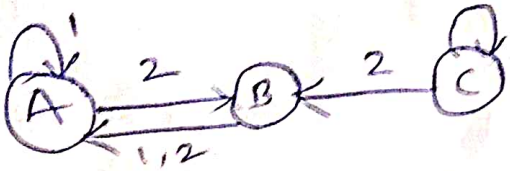
2) For every state & input there is exactly one transition specified to exactly one. possibly the same state

3) For every transition there is one output action specified that output could be trivial, but atleast one output does something sensible.

4) For every state there is a sequence of inputs that the system back to the same state.

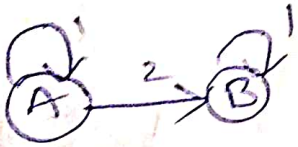


The state B can never be left. The initial state can never be entered again

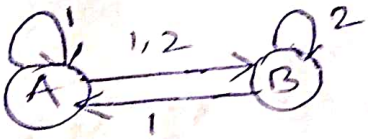


State C cannot be entered

State A and B are not reachable



No transition is specified for an input of 2 when in state A



Two transitions are specified for an input of 1 in state A

→ There are 2 aspects of state graphs

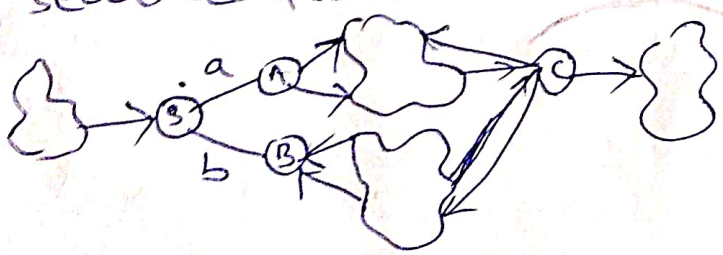
1) The states with their transitions & ~~outp~~ inputs that cause them

2) outputs associated with the transition

→ two state graphs with identical states, inputs & transitions would have different outputs.

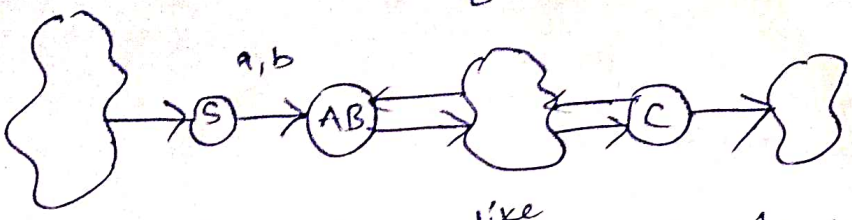
Equivalent states

Two states are equivalent if every sequence of inputs starting from one state produces exactly the same sequence of outputs when started from other state



⇒ Equivalent states

equivalent states of merged.



Unreachable state: Is like unreachable code - a state that no input sequence can reach

→ An unreachable state is not impossible.

There are 2 possibilities

1) There is a bug that is some transitions are missing
2) The transitions are there but you don't know about it.

Dead state: Is a state that once entered can't be left

→ A set of states may appear to be dead because the program has 2 modes of operation. In first mode it goes through initialization process that consists of several states, once initialized it goes strongly connected states of working states which with in the context of routine which can not be exited.

→ The initialization states are unreachable to the working states & the working states are to be dead to the initialization states

→ The second mode is the only way to get back from dead state is either system crash (or) restart.

→ The states, transitions & the inputs could be correct there could be no dead (or) unreachable state but the output for the transition could be incorrect

→ The behaviour of finite state machine is invariant under all encodings. That's why the states are numbered from 1 to n.

State Testing

Impact of Bugs : A bug can itself as one (or) more of the following symptoms.

1) Wrong number of states
2) Wrong transition for a given state-input combination
3) Wrong output for a given transition
4) set of states that are inadvertently made equivalent (accidentally)
5) set of states that are split to create inequivalent duplicates
6) set of states that have become dead
7) set of states that have become unreachable

Principles

→ A path in a state graph is a succession of transition caused by sequence of inputs

→ The starting point of state testing is to define a set of covering input sequences that get back to the initial state.

→ For each step in each input sequence, define the expected next state, expected transitions & expect o/p.

→ set of tests consists of 3 sets of sequences

1) Input sequences
2) Corresponding-transitions (or) next state names
3) output sequences

Limitations & Extensions

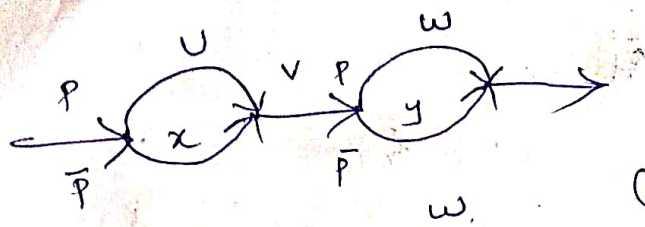
1) simply identifying the factors that contribute to the state, calculating total number of states & comparing this number to designer's notion catches some bugs.
2) Insisting on a justification for all & dead, unreachable, & impossible states & transitions catches few more bugs.
3) Insisting on an explicit specification of transition & output for every combination of input & state catches ~~may~~ many more bugs.
4) A set of input sequences that provide coverage of all nodes & links is a mandatory minimum requirement.
5) In executing state tests, it is essential that means be provided to record sequence of states resulting from the input sequence & outputs that result from input sequence.

Testability Tips : Is a test the software

→ The key to testability design is easy, build explicit finite state machines
→ There are about 80 possible good & bad 3 state machines, 2700 4-state machines, 2,75,000 five state machines & close to 100 million six-state machines most of which are bad.
→ create finite state machine models & identify how you are implementing every part of model with 4 (or) 5 states.

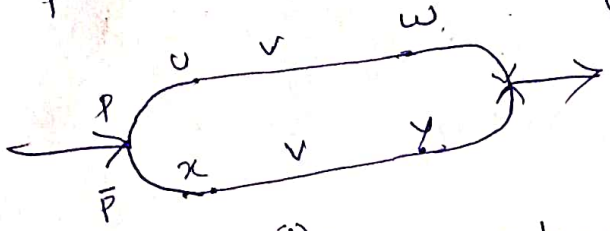
→ Switches, flags & unachievable paths

(a)



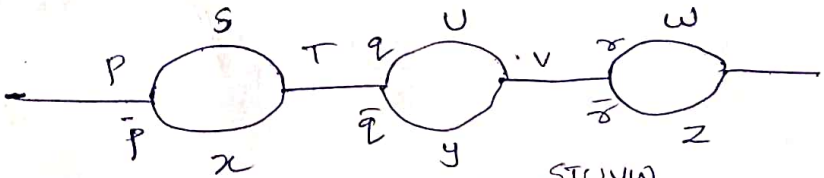
(a) (program with one switch variable)

(b)

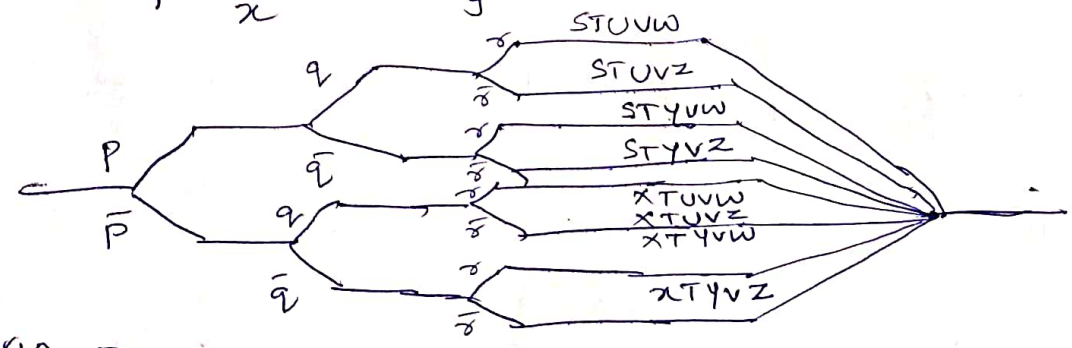


→ Above figure (b) ~~show~~ we have rewritten the routine to eliminate the flag $(P, \bar{P})$ as soon as the flag value is calculated we branch the cost is the cost of converting segment V into a subroutine & calling it twice.

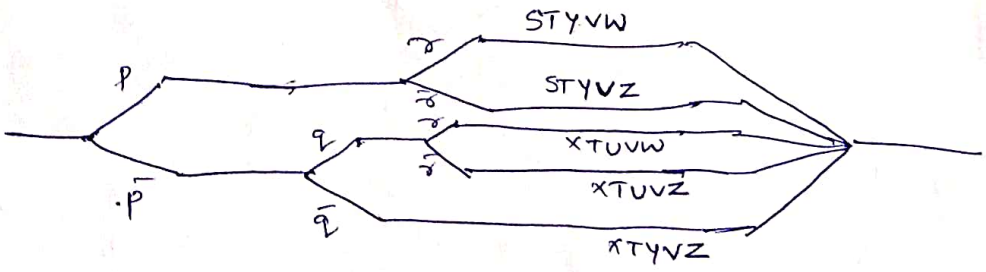
→ we have 4 paths, two of which are achievable, two of which are achievable & needed to achieve branch coverage.



(a) original program



(b) Expanded to Remove state Machine Behaviour



(c) Pruned to Remove Unwanted Branches

↑ Three switch variable program

→ The fig" is complicated because there are 3 variables we can put the decision up front & branch directly & again we use subroutines to make each path explicitly & do with the switches.

→ The advantage of this implementation is that if any of the combinations are not needed we dipout that part.

→ The fig(c) the paths are achievable & all paths are needed for branch cover.

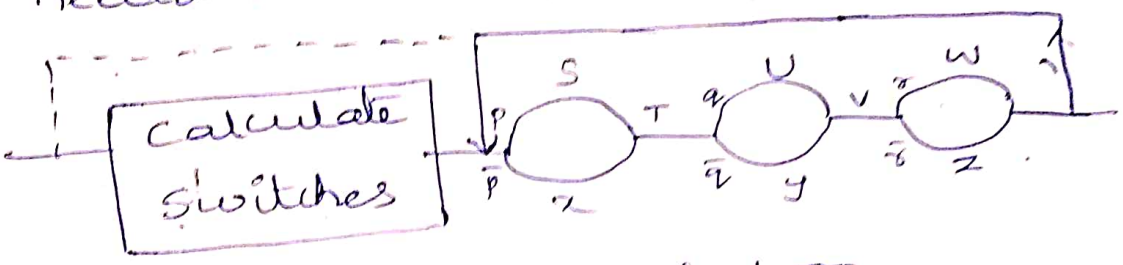


fig: Switches in Loop

→ above fig" is similar to previous but we have included switched part in loop. we don't know which paths are achievable & which are not required it depends on the switch settings, we must do a attempt branch coverage in every possible state.

→ The simplest essential finite state machine is a flipflop. There is no logic that can implement it without feedback we can't describe the behavior of machine by a decision table unless you provide feedback into table (or) call it recursively.

Design Guidelines

1) Start by designing the abstract machine. verify that it is what you want to do. Do an explicit analysis in the form of state graph (or) table for anything with 3 states

2) Start with an explicit design i.e. input encoding output encoding, state code assignment, transition table, output table, state storage, & how you intend to form the state symbol product.

3) Before you start, see it really matters, neither the time nor memory for the explicit implementation, usually matters

4) Take shortcuts by making things implicit only as you must to make significant reductions in time (or) space in the context of whole system

5) The order in which you should make things implicit are output encoding, input encoding, state code, state symbol product, output table, transition table, state storage. That's the order from least to most dangerous.

6) Consider a hierarchical design if you have more than dozen states.

7) Build, buy (or) implement tools languages that implement finite state machines as software.

8) Build in the means to initialize to any state.
Build " " transition verification instrumentation.
These are much easier to do with an explicit machine