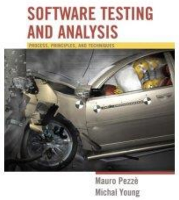
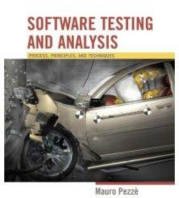
Program Analysis



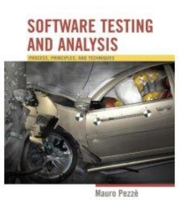
Why Analysis

- Exhaustively check properties that are difficult to test
  - Faults that cause failures
    - rarely
    - under conditions difficult to control
  - Examples
    - race conditions
    - faulty memory accesses
- Extract and summarize information for inspection and test design



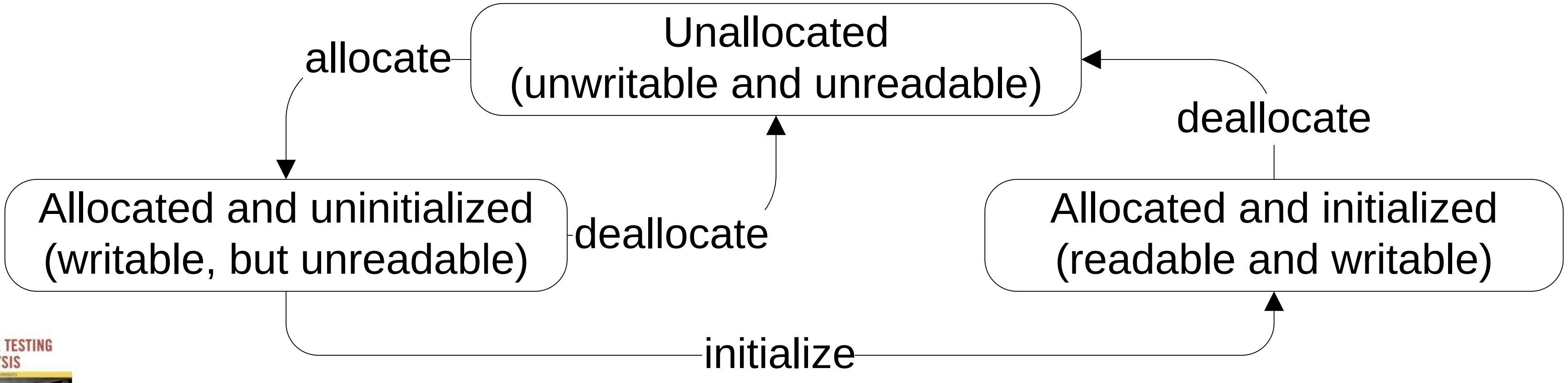
Why automated analysis

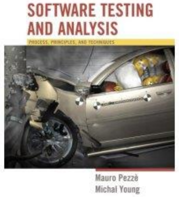
- Manual program inspection
  - effective in finding faults difficult to detect with testing
  - But humans are not good at
    - repetitive and tedious tasks
    - maintaining large amounts of detail
- Automated analysis
  - replace human inspection for some class of faults
  - support inspection by
    - automating extracting and summarizing information
    - navigating through relevant information



Memory Analysis

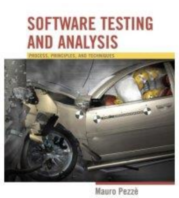
- Instrument program to trace memory access
  - record the state of each memory location
  - detect accesses incompatible with the current state
    - attempts to access unallocated memory
    - read from uninitialized memory locations
  - array bounds violations:
    - add memory locations with state *unallocated* before and after each array
    - attempts to access these locations are detected immediately





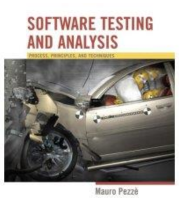
Data Races

- Testing: not effective
  (nondeterministic interleaving of threads)
- Static analysis:
  computationally expensive, and approximated
- Dynamic analysis:
  can amplify sensitivity of testing to detect potential data races
  - avoid pessimistic inaccuracy of finite state verification
  - Reduce optimistic inaccuracy of testing



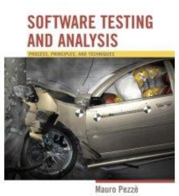
Dynamic Lockset Analysis

- Lockset discipline: set of rules to prevent data races
  - Every variable shared between threads must be protected by a mutual exclusion lock
  - ….
- Dynamic lockset analysis detects violation of the locking discipline
  - Identify set of mutual exclusion locks held by threads when accessing each shared variable
  - INIT: each shared variable is associated with all available locks
  - RUN: thread accesses a shared variable
    - intersect current set of candidate locks with locks held by the thread
  - END: set of locks after executing a test = set of locks always held by threads accessing that variable
    - empty set for v = no lock consistently protects v



Simple lockset analysis: example

Thread	Program trace	Locks held	Lockset(x)	
		{}	{lck1, lck2}	INIT:all locks for x
thread A	lock(lck1)			
		{lck1}		lck1 held
	x=x+1			
			{lck1}	Intersect with locks held
	unlock(lck1}			
		{}		
tread B	lock{lck2}			
		{lck2}		lck2 held
	x=x+1			
			{}	Empty intersection potential race
	unlock(lck2}			
		{}		



Predicate templates

over one variable	
constant	x=a
uninitialized	x=uninit
small value set	x={a,b,c}
over a single	*numeric variable*
in a range	$x \geq a, x \leq b, a \leq x \leq b$
nonzero	$x \neq 0$
modulus	x=a(mod b)
nonmodulus	$x \neq$ a(mod b)
over the sum of	*two numeric variables*
linear relationship	y=ax+b
ordering relationship	$x \leq y, x<y, x=y, x \neq y$
…