Controlling Risks
Safety System Models

Module A

Software Reliability

- Software does not wear out and there are no latent manufacturing defects.
  - However, there may be bugs
    - Due to programmer error
    - Or poor requirements specification

Software Reliability

- Software reliability is the ability of the software to perform the expected function when needed.
- As software becomes more complex the ability to verify correctness increases exponentially.

$$Verification = O^n$$

$$\lim_{n \to \infty} \left(\frac{time}{function} \right)^n$$

Stress-Strength

- Software strength is affected by the amount of stress-rejection designed into the software.
- Software that checks for valid inputs and rejects invalid inputs will fail much less frequently.
- The stress to a software system is the combination of inputs, timing of inputs and stored data seen by the CPU.

Software Diagnostics

- Software diagnostics and stress rejection increase software strength.
- Software Diagnostics
  - Automatic software verification during execution
  - Prevents software failures
  - Identifies faults

Stress Rejection

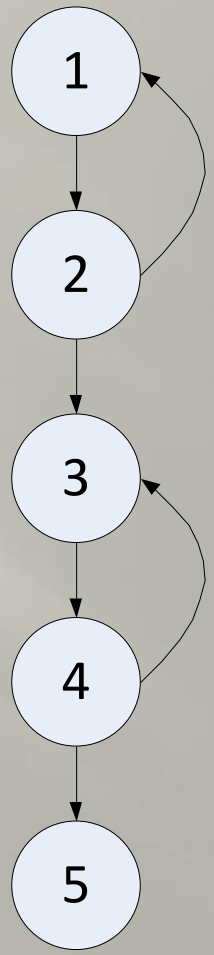
- Potential stressors that might cause software failure are filtered.
- Plausibility assertions check the inputs to software and stored data.
- Data format and range is checked before commands are executed.
- Data pointers are verified to be within a valid range for an array.

McCabe Complexity

- The number of control flow paths in an algorithm may be calculated using the MaCabe Complexity Metric

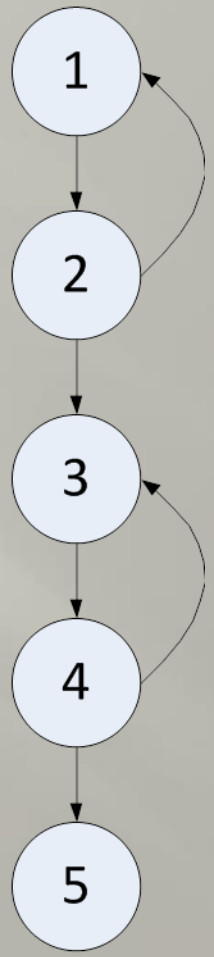
$$NP = e - n + 2$$

NP = 6 – 5 + 2 = 3



Control Flow Testing

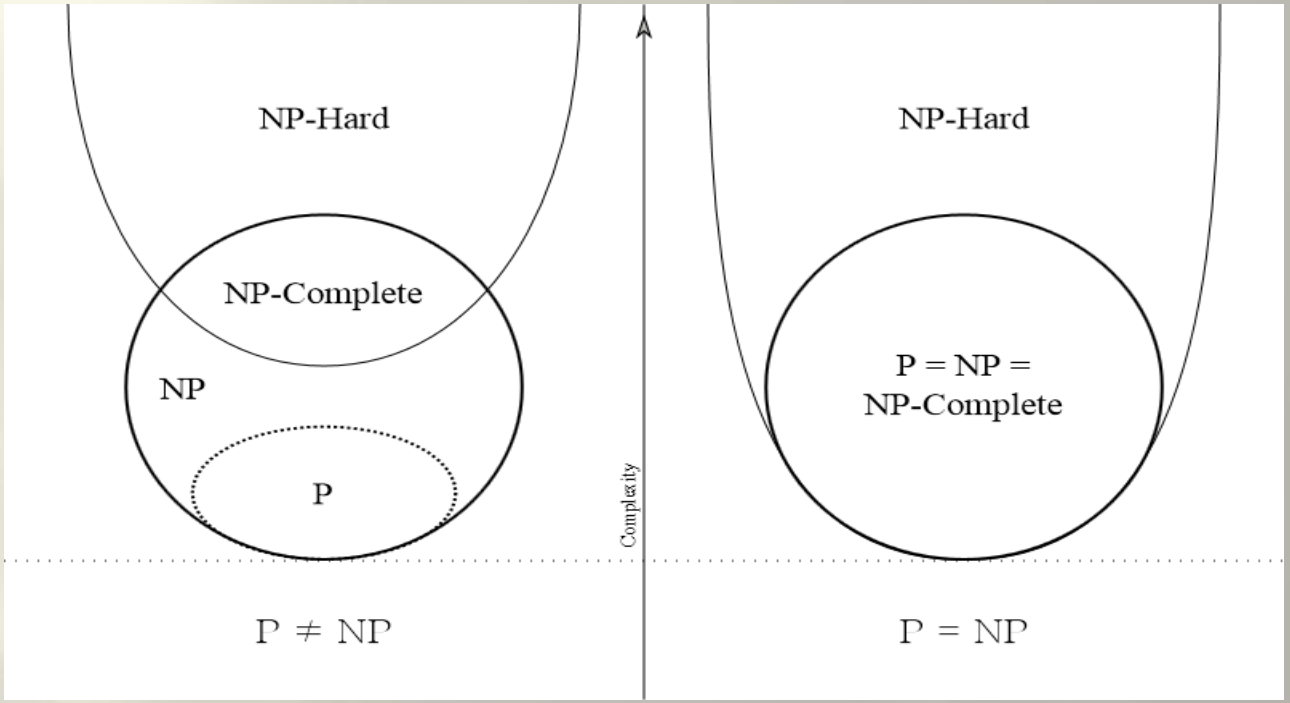
- The number of paths in the algorithm provides the number tests that must be executed to verify the correctness of the program.
  - Does not account for path variations due to input data.
  - Testing all paths may not detect all software design faults.



NP-complete

- A decision problem *L* is NP-complete if it is in the set of NP problems so that any given solution to the decision problem can be verified in polynomial time, and also in the set of NP-hard problems so that any NP problem can be converted into *L* by a transformation of the inputs in polynomial time.
- The most notable characteristic of NP-complete problems is that no fast solution to them is known. That is, the time required to solve the problem using any currently known algorithm increases very quickly as the size of the problem grows.

An Exercise Left to the Student



Determining whether or not it is possible to solve these problems quickly, called the P versus NP problem, is one of the principal unsolved problems in computer science today.

The Clay Mathematics Institute is offering a $1 million reward to anyone who has a formal proof that P=NP or that P≠NP.

Polynomial time

- An algorithm is said to be of **polynomial time** if its running time is upper bounded by a polynomial expression in the size of the input for the algorithm, i.e., $T(n) = O(n^k)$ for some constant k. Problems for which a polynomial time algorithm exists belong to the complexity class **P**.
  - The quicksort sorting algorithm on n integers performs at most An^2 operations for some constant A. Thus it runs in time $O(n^2)$ and is a polynomial time algorithm.
  - All the basic arithmetic operations (addition, subtraction, multiplication, division, and comparison) can be done in polynomial time.

Code Trace Verification

- The use of personnel to determine the correctness of software code does not fall in polynomial time.
- This is because the application software is designed and compiled using the SDK (software development kit), running on an OS, that will run on a PLC.
- However, code trace verification (checking code line-by-line) is very useful and should be performed.
  - Realize that it does not account for 100% complete verification of correctness.

Input Space

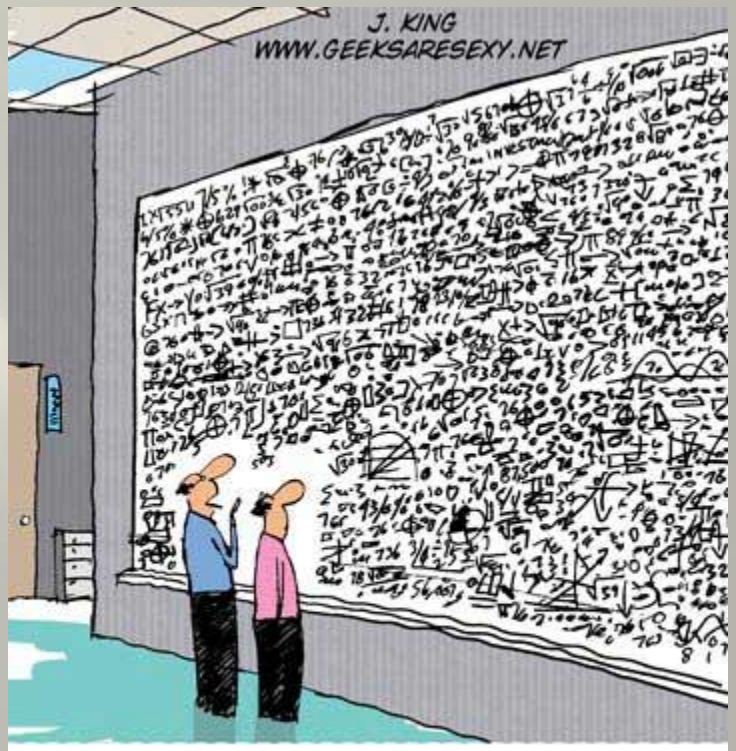
- The input space is the collection of all possible input conditions or sequences of input conditions.
- The input space view of program operation offers an advantage in that program execution paths can be estimated in terms of the functions being performed.

Software Modeling

- If the number of possible execution sequences is very large, then software can be modeled statistically.

Basic Model

- Assume that there are some number of software design faults.
- All faults are likely to cause failure and be repaired.
- The failure rate is proportional to the current number of faults in the program.



Basic Model

- N_0 – the number of faults at the beginning of the test period.
- $N_c(t)$ – the number of repaired faults.
- k – the ratio of remaining faults and field failure rate per hour.

$$\lambda(n_c) = k[N_0 - n_c(t)]$$

$$\lim_{t \to \infty} k[N_0 - n_c(t)] = 0$$

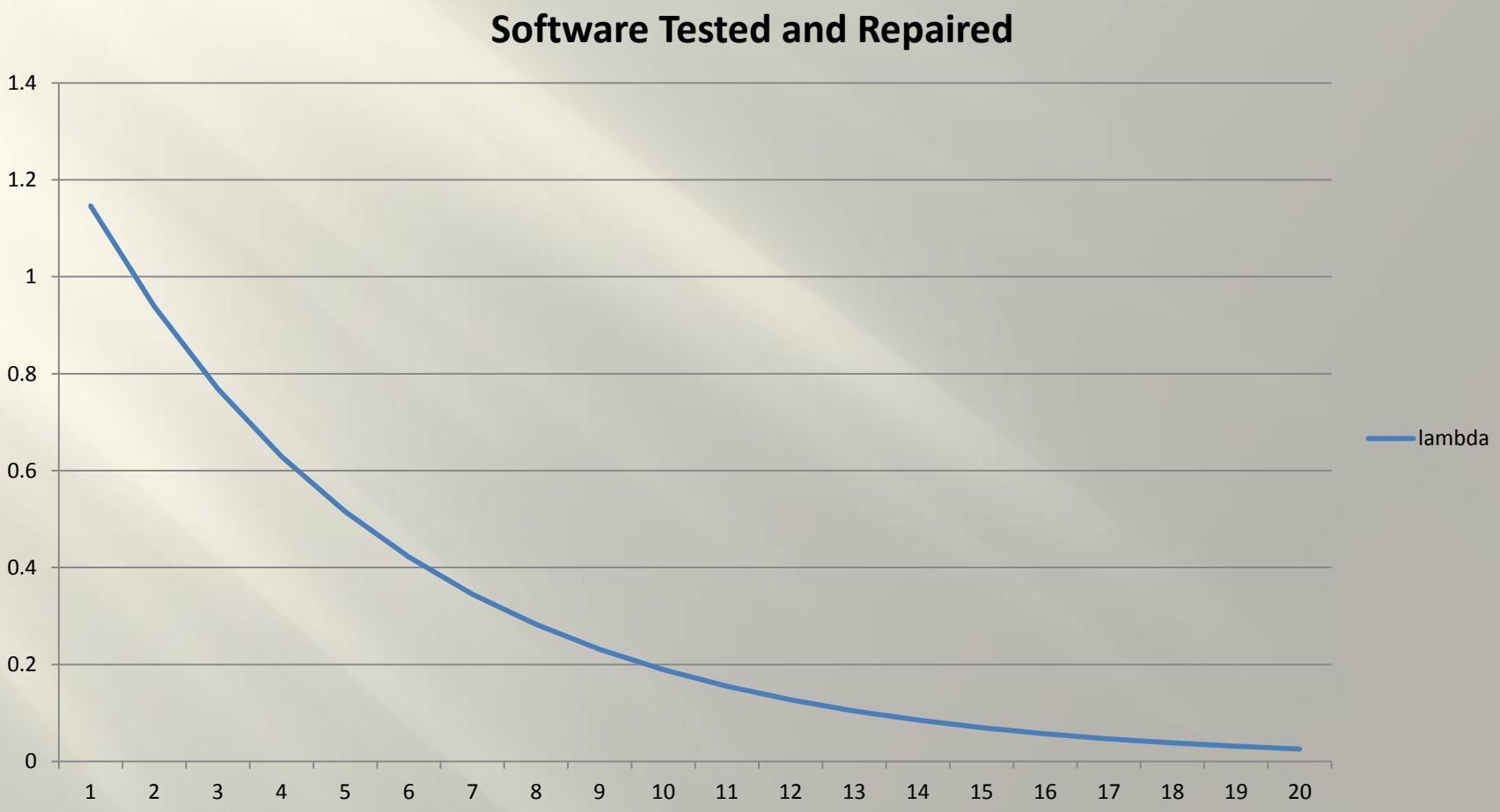
Non-linear Repair Rate

- We must assume that the actual repair does not occur in linear time, since software faults are not repaired at a constant rate.
- A closer approximation is

$$\lambda(\tau) = k\, N_0 e^{-k\tau}$$

- The failure rate is an exponentially decreasing function with time.

Non-linear Failure Rate



Software Reliability Model Assumptions

- Faults are independent
  - Faults are usually introduced from misunderstood functional requirements, design error, coding error
  - These usually result in independent faults
- Times Between Failures is independent
  - When testing follows a plan, failures cause more intensified verification in the fault area
  - This assumption is not valid for most testing processes

Software Reliability Model Assumptions

- Detected faults are removed in negligible time
  - This assumption is almost always violated in real projects
- No new faults are introduced
  - Yeah, right
- Faults are equal
  - Some faults are found quickly and other may exist in software for a long time before the fault is found.
  - The assumption is reasonable because initial testing finds the obvious faults in the beginning of the failure rate curve

Summary

- Here's what you should walk away with
  - Programmable safety systems reject many stressors
  - Diagnostics improve the ability to detect faults
  - The complexity of verifying algorithms increases with the lines of code
  - The time spent testing the code directly affects the failure rate of the software application
- My advice
  - Review the code after significant effort has been made to test the algorithm!