Software System Safety

Nancy G. Leveson

MIT Aero/Astro Dept. (leveson@mit.edu)
http://sunnyday.mit.edu
MIT, Room 33−334
77 Massachusetts Ave., Cambridge MA 02139
Tel: 617−258−0505

Overview of the Class

Week 1: Understanding the Problem

Week 2: The Overall Process and Tasks

Week 3: Hazard Causal Analysis (Root Cause Analysis)

Week 4: A New Approach to Hazard, Root Cause, and Accident Analysis

Week 5: Requirements Analysis

Week 6: Design for Safety

Week 7: Human−Machine Interaction and Safety

Week 8: Testing and Assurance, Operations, Maintenance

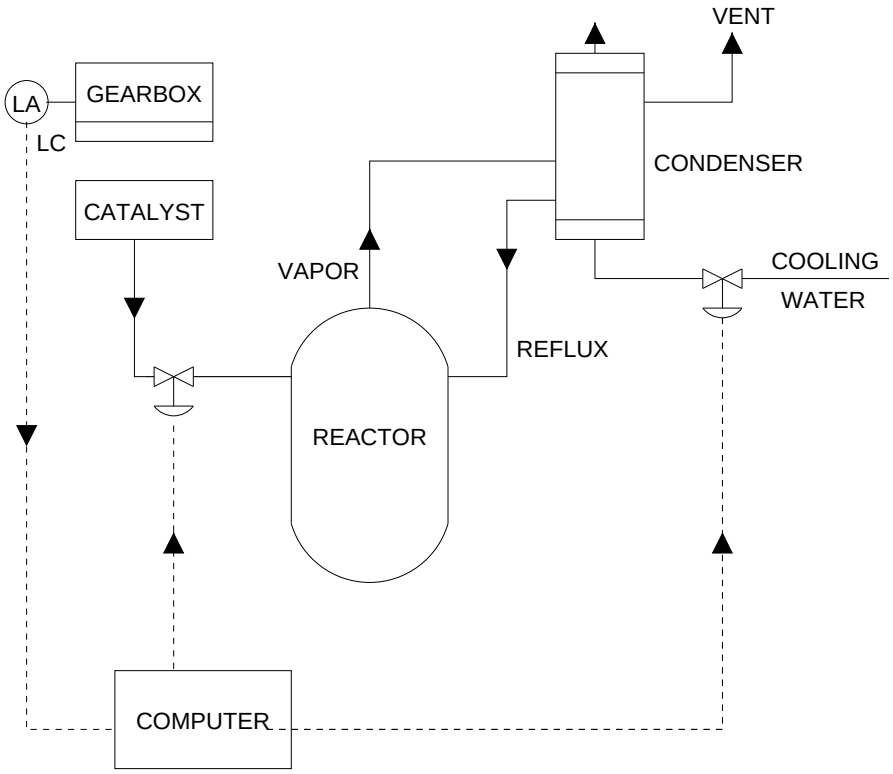
Week 9: Management and Organizational Issues (including safety culture)

Week 10: Summary and Conclusions

The Problem

The first step in solving any problem is to understand it. We often propose solutions to problems that we do not understand and then are surprised when the solutions fail to have the anticipated effect.

Accident with No Component Failures



Types of Accidents

- Component Failure Accidents
  - Single or multiple component failures
  - Usually assume random failure
- System Accidents
  - Arise in interactions among components
    - No components may have "failed"
  - Caused by interactive complexity and tight coupling
  - Exacerbated by the introduction of computers.

Interactive Complexity

- Complexity is a moving target
- The underlying factor is intellectual manageability

1. A "simple" system has a small number of unknowns in its interactions within the system and with its environment.
2. A system is intellectually unmanageable when the level of interactions reaches the point where they cannot be thoroughly
   - planned
   - understood
   - anticipated
   - guarded against
3. Introducing new technology introduces unknowns and even "unk−unks."

Computers and Risk

We seem not to trust one another as much as would be desirable. In lieu of trusting each other, are we putting too much trust in our technology? . . . Perhaps we are not educating our children sufficiently well to understand the reasonable uses and limits of technology.

Thomas B. Sheridan

The Computer Revolution

- Software is simply the design of a machine abstracted from its physical realization.
- Machines that were physically impossible or impractical to build become feasible.
- Design can be changed without retooling or manufacturing.
- Can concentrate on steps to be achieved without worrying about how steps will be realized physically.

Advantages = Disadvantages

- Computer so powerful and so useful because it has eliminated many of physical constraints of previous machines.
- Both its blessing and its curse:
  - \+ No longer have to worry about physical realization of our designs.
  - − No longer have physical laws that limit the complexity of our designs.

The Curse of Flexibility

- Software is the resting place of afterthoughts
- No physical constraints
  - To enforce discipline on design, construction and modification
  - To control complexity
- So flexible that start working with it before fully understanding what need to do
- ‘‘And they looked upon the software and saw that it was good, but they just had to add one other feature ...’’

Software Myths

1. Good software engineering is the same for all types of software.
2. Software is easy to change.
3. Software errors are simply ‘‘teething’’ problems.
4. Reusing software will increase safety.
5. Testing or ‘‘proving’’ software correct will remove all the errors.

Black Box Testing

Test data derived solely from specification (i.e., without knowledge of internal structure of program).

- Need to test every possible input

  x := y * 2

  (since black box, only way to be sure to detect this is to try every input condition)

  - Valid inputs up to max size of machine (not astronomical)
  - Also all invalid input (e.g., testing Ada compiler requires all valid and invalid programs)
  - If program has ‘‘memory’’, need to test all possible unique valid and invalid sequences.

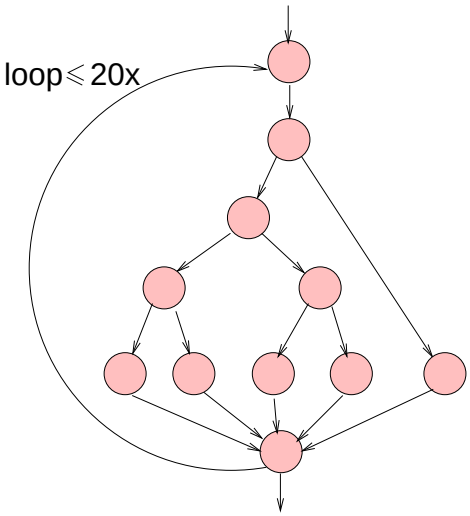
- So for most programs, exhaustive in put testing is impractical.

White Box Testing

Derive test data by examining program’s logic.

Exhaustic path testing: Two flaws

1) Number of unique paths through program is astronomical.



(control−flow graph)

$$5^{20} + 5^{19} + 5^{18} + \ldots + 5 = 10^{14}$$

$$= 100 \text{ trillion}$$

If could develop/execute/verify one test case every five minutes = 1 billion years

If had magic test processor that could develop/execute/evaluate one test per msec = 3170 years.

White Box Testing (con't)

2) Could test every path and program may still have errors!

- Does not guarantee program matches specification, i.e., wrong program.
- Missing paths: would not detect absence of necessary paths
- Could still have data−sensitivity errors.

  e.g. program has to compare two numbers for convergence

  if (A − B) < epsilon ...

  is wrong because should compare to abs(A − B)

  Detection of this error dependent on values used for A and B and would not necessarily be found by executing every path through program.

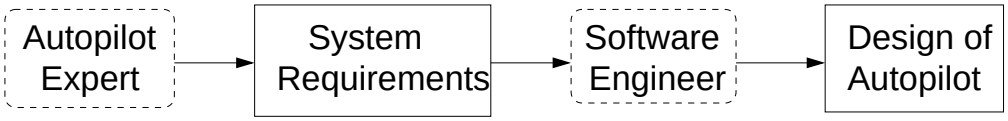
Mathematical Modeling Difficulties

- Large number of states and lack of regularity
- Lack of physical continuity: requires discrete rather than continuous math
- Specifications and proofs using logic:
  - May be same size or larger than code
  - More difficult to construct than code
  - Harder to understand than code

  Therefore, as difficult and error−prone as code itself
- Have not found good ways to measure software quality

Abstraction from Physical Design

- Software engineers are doing system design



- Most errors in operational software related to requirements
  - Completeness a particular problem
- Software "failure modes" are different
  - Usually does exactly what you tell it to do
  - Problems occur from operation, not lack of operation
  - Usually doing exactly what software engineers wanted

Ways to Cope with Complexity

- Analytic Reduction (Descartes)
  - Divide system into distinct parts for analysis purposes.
  - Examine the parts separately.

- Three important assumptions:

1. The division into parts will not distort the phenomenon being studied.
2. Components are the same when examined singly as when playing their part in the whole.
3. Principles governing the assembling of the components into the whole are themselves straightforward.

Ways to Cope with Complexity (con't.)

- Statistics
  - Treat as a structureless mass with interchangeable parts.
  - Use Law of Large Numbers to describe behavior in terms of averages.
- Assumes components sufficiently regular and random in their behavior that they can be studied statistically.

What about software?

- Too complex for complete analysis:
  - Separation into non−interacting subsystems distorts the results.
  - The most important properties are emergent.
- Too organized for statistics
  - Too much underlying structure that distorts the statistics.

Systems Theory

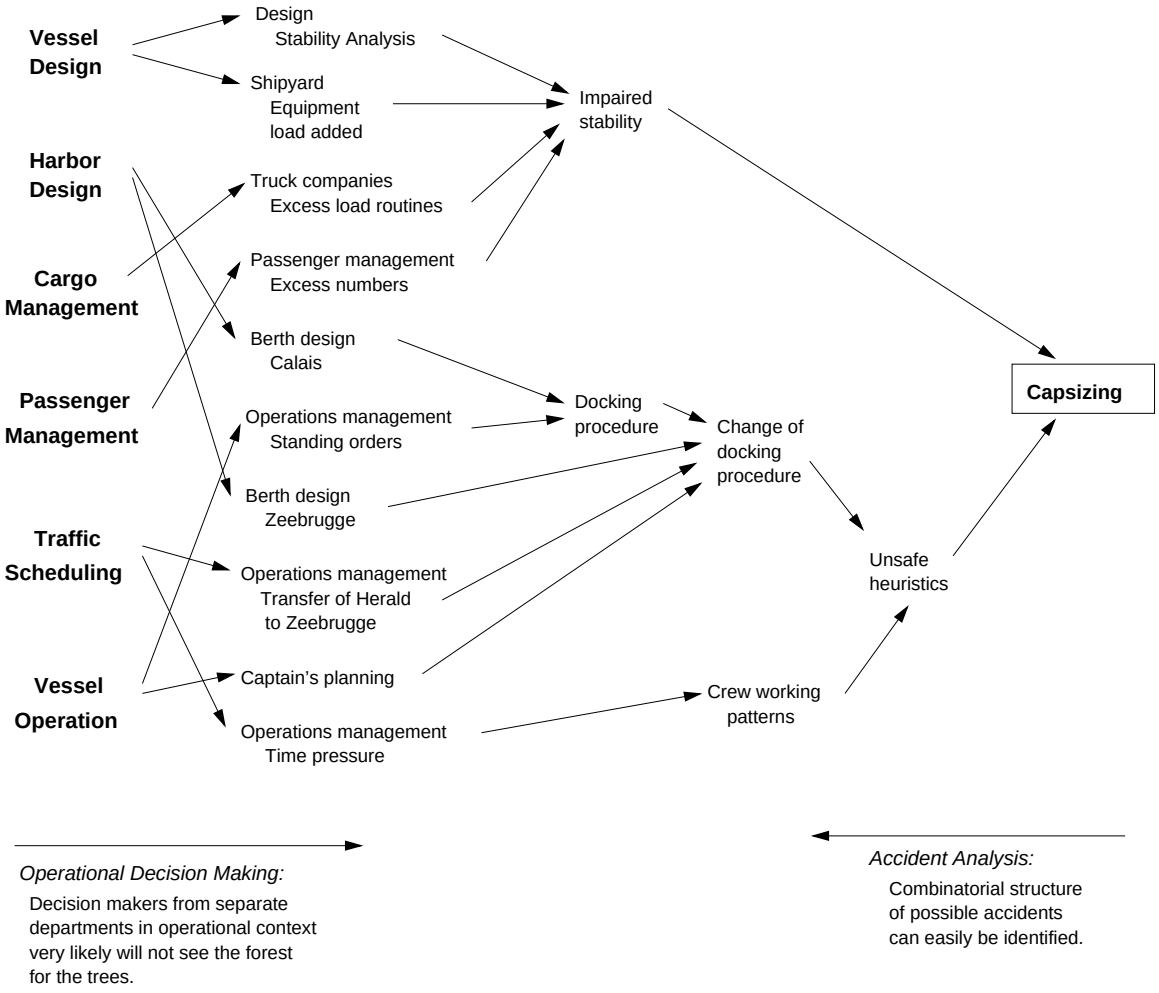
- Developed for biology (Bertalanffly) and cybernetics (Norbert Weiner)
  - For systems too complex for complete analysis
    - Separation into non−interacting subsystems distorts results
    - Most important properties are emergent.

    and too organized for statistical analysis

- Concentrates on analysis and design of whole as distinct from parts (basis of system engineering)
  - Some properties can only be treated adequately in their entirety, taking into account all social and technical aspects.
  - These properties derive from relationships between the parts of systems −− how they interact and fit together.

Systems Theory (2)

- Two pairs of ideas:
  1. Emergence and hierarchy
     - Levels of organization, each more complex than one below.
     - Levels characterized by emergent properties
       - Irreducible
       - Represent constraints upon the degree of freedom of components a lower level.
     - Safety is an emergent system property
       - It is NOT a component property.
       - It can only be analyzed in the context of the whole.



Systems Theory (3)

2. Communication and control
   - Hierarchies characterized by control processes working at the interfaces between levels.
   - A control action imposes constraints upon the activity at a lower level of the hierarchy.
   - Open systems are viewed as interrelated components kept in a state of dynamic equilibrium by feedback loops of information and control.
   - Control in open systems implies need for communication

Stages in Process Control System Evolution

1. Mechanical systems
   - Direct sensory perception of process
   - Displays are directly connected to process and thus are physical extensions of it.
   - Design decisions highly constrained by:
     - Available space
     - Physics of underlying process
     - Limited possibility of action at a distance

Stages in Process Control System Evolution (2)

2. Electromechanical systems
   - Capability for action at a distance
   - Need to provide an image of process to operators
   - Need to provide feedback on actions taken.
   - Relaxed constraints on designers but created new possibilities for designer and operator error.

Stages in Process Control System Evolution (3)

3. Computer−based systems
   - Allow multiplexing of controls and displays.
   - Relaxes even more constraints and introduces more possibility for error.
   - But constraints shaped environment in ways that efficiently transmitted valuable process information and supported cognitive processes of operators.
   - Finding it hard to capture and present these qualities in new systems.

A Possible Solution

- Enforce discipline and control complexity
  - Limits have changed from structural integrity and physical constraints of materials to intellectual limits
- Improve communication among engineers
- Build safety in by enforcing constraints on behavior

Control software contributes to accidents by:
1. Not enforcing constraints on behavior
2. Commanding behavior that violates constraints

Example (batch reactor)

System safety constraint:

Water must be flowing into reflux condenser whenever catalyst is added to reactor.

Software safety constraint:

Software must always open water valve before catalyst valve

The Problem to be Solved

- The primary safety problem in software−intensive systems is the lack of appropriate constraints on design.
- The job of the system safety engineer is to identify the design constraints necessary to maintain safety and to ensure the system and software design enforces them.

Safety ≠ Reliability

Accidents in high−tech systems are changing their nature, and we must change our approaches to safety accordingly.

Confusing Safety and Reliability

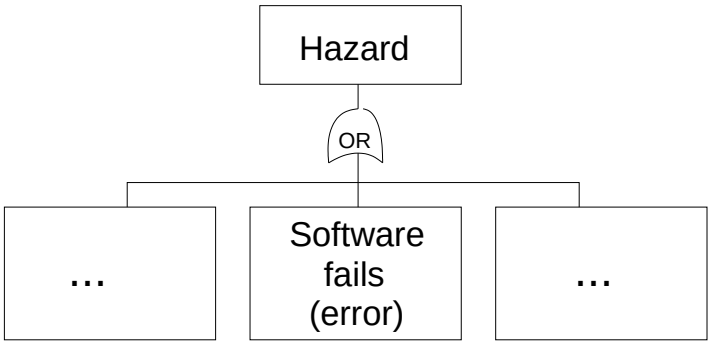
From an FAA report on ATC software architectures:

> "The FAA's en route automation meets the criteria for consideration as a safety−critical system. Therefore, en route automation systems must posses ultra−high reliability."

From a blue ribbon panel report on the V−22 Osprey problems:

> "Safety [software]: ...
>
> Recommendation: Improve reliability, then verify by extensive test/fix/test in challenging environments."

Typical Fault Trees



Hazard Cause	Probability	Mitigation
Software Error	0	Test software

What is a Software "Failure"?

Failure: Nonperformance or inability of system or component to perform its intended function for a specified time under specified environmental conditions.

A basic abnormal occurrence, e.g.,

- burned out bearing in a pump
- relay not closing properly when voltage applied

Fault: Higher−order events, e.g.,

- relay closes at wrong time due to improper functioning of an upstream component.

All failures are faults but not all faults are failures.

Reliability Engineering Approach to Safety

Reliability: The probability an item will perform its required function in the specified manner over a given time period and under specified or assumed conditions.

(Note: Most software−related accidents result from errors in specified requirements or function and deviations from assumed conditions.)

- Concerned primarily with failures and failure rate reduction
  - Parallel redundancy
  - Standby sparing
  - Safety factors and margins
  - Derating
  - Screening
  - Timed replacements

Reliability Engineering Approach to Safety (2)

- Assumes accidents are the result of component failure.
  - + Techniques exist to increase component reliability
    Failure rates in hardware are quantifiable.
  - – Omits important factors in accidents.
    May even decrease safety.
- Many accidents occur without any component ‘‘failure’’

  e.g. Accidents may be caused by equipment operation outside parameters and time limits upon which reliability analyses are based.

  Or may be caused by interactions of components all operating according to specification

Highly reliable components are not necessarily safe.

Reliability Approach to Software Safety

Standard engineering techniques of

- Preventing failures through redundancy
- Increasing component reliability
- Reuse of designs and learning from experience

won’t work for software and system accidents.

Preventing Failures through Redundancy

- Redundancy simply makes complexity worse.
  - NASA experimental aircraft example
  - Any solutions that involve adding complexity will not not solve problems that stem from intellectual unmanageability and interactive complexity.
- Majority of software−related accidents caused by requirements errors.
- Does not work for software even if accident is caused by a software implementation error.

  Software errors not caused by random wearout failures.

Increasing Software Reliability (Integrity)

- Appearing in many new international standards for software safety (e.g., 61508)
  - "Safety integrity level"
  - Sometimes give reliability number (e.g., 10^{-9})

    Can software reliability be measured? What does it even mean?
- Safety involves more than simply getting software "correct"

  Example: altitude switch

  1. Signal safety−increasing =>
     Require any of three altimeters report below threshold
  2. Signal safety−reducing =>
     Require all three altimeters to report below threshold

Software Component Reuse

- One of most common factors in software−related accidents
- Software contains assumptions about its environment.

  Accidents occur when these assumptions are incorrect.

  - Therac−25
  - Ariane 5
  - U.K. ATC software
- Most likely to change the features embedded in or controlled by the software.
- COTS makes safety analysis more difficult.

Safety and reliability are different qualities!

Software−Related Accidents

- Are usually caused by flawed requirements
  - Incomplete or wrong assumptions about operation of controlled system or required operation of computer.
  - Unhandled controlled−system states and environmental conditions.
- Merely trying to get the software ‘‘correct’’ or to make it reliable will not make it safer under these conditions.

Software−Related Accidents (con't.)

- Software may be highly reliable and ''correct'' and still be unsafe.
  - Correctly implements requirements but specified behavior unsafe from a system perspective.
  - Requirements do not specify some particular behavior required for system safety (incomplete)
  - Software has unintended (and unsafe) behavior beyond what is specified in requirements.

Systemic Factors in (Software−Related) Accidents

1. Flaws in the Safety Culture

Safety Culture: The general attitude and approach to safety reflected by those who participate in an industry or organization, including management, workers, and government regulators

- Underestimating or not understanding software risks
- Overconfidence and complacency
- Assuming risk decreases over time
- Ignoring warning signs
- Inadequate emphasis on risk management
- Incorrect prioritization of changes to automation
- Slow understanding of problems in human−automation mismatch
- Overrelying on redundancy and protection systems
- Unrealistic risk assessment

Systemic Factors (con't)

2. Organizational Structure and Communication
   - Diffusion of responsibility and authority
   - Limited communication channels and poor information flow

3. Technical Activities
   - Flawed review process
   - Inadequate specifications and requirements validation
   - Flawed or inadequate analysis of software functions
   - Violation of basic safety engineering practices in digital components
   - Inadequate system engineering
   - Lack of defensive programming
   - Software reuse without appropriate safety analysis

Systemic Factors (con't)

- Inadequate system safety engineering
- Unnecessary complexity and software functions
- Test and simulation environment does not match operations
- Deficiencies in safety−related information collection and use
- Operational personnel not understanding automation
- Inadequate design of feedback to operators
- Conflicting and inadequate documentation for operators
- Inadequate cognitive engineering